Algoritmos de Ordenação

Daniel de Souza Rodrigues - 18.2.8112 - Sistemas de Informação

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
Rua 36, Número 115 - Bairro Loanda, João Monlevade - CEP: 35931-008

daniel.sr@aluno.ufop.edu.br

Abstract. *In this article the following sorting algorithms will be presented: Selection Sort, Sort Insertion, Bubble Sort, Sort Sort, Quick Sort, Heap Sort, Merge Sort; will also be presented the way in which each algorithm behaves; analysis of time and memory complexity; results of the worst and best case of the algorithms; a brief citation on the average case of each algorithm; negative and positive points about the use of each algorithm and their respective classifications.*

Resumo. *Neste artigo serão apresentados os seguintes algoritmos de ordenação: Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Quick Sort, Heap Sort, Merge Sort; também será apresentado o modo em que cada algoritmo se comporta; analise de complexibilidade de tempo e memoria; resultados do pior e melhor caso dos algoritmos; pontos negativos e positivos sobre a utilização de cada algoritmo e suas respectivas classificações.*

1. Introdução

O artigo apresenta os algoritmos de ordenação; Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Quick Sort, Heap Sort, Merge Sort e seus respectivos codigo em linguagem de programação C, todos requisitos de Atribuiçoes(Swap) e Comparações feitas(If) mediante as entradas(arrays) determinadas estão documentados na seção de Testes Realizados $(.3)$ de cada Seção.

Considerações:

Todos os testes foram executados com Processador: $AMD - Ryzen3 - 2200g$ e Memoróriа: $Ballistix$ - $8GB\ DDR4\ 2400mhz$

2. Bubble Sort

Este algoritmo é um dos mais simples levando em conta a implementação, apesar de não ter um bom desempenho se comparado a outros tipos de algoritmos de ordenação.[Wikipédia 2019a]

```c
void swap(float *elemento1,float *elemento2){

    float aux = *elemento1;
                *elemento1 = *elemento2;
                *elemento2 = aux;
}
void bubble_Sort(float *vetor,int tam){

    for(int i=0; i<tam-1;i++){

        for(int j=0;j<tam-i-1;j++){

            if(vetor[j]>vetor[j+1])
            {
               swap(&vetor[j],&vetor[j+1]);
            }
        }
    }
}
```

Figure 1. Bubble Sort em C

2.1. Implementação

A ideia é sempre iterar toda a lista de itens quantas vezes forem necessário até que os itens estejam na ordem correta, durante a iteração, o algoritmo deve comparar e ordenar pares de valores: O item que se está iterando e o seu vizinho à direita.

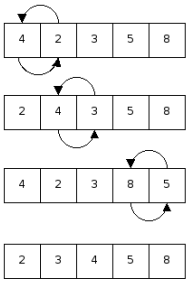

Figure 2. Bubble Sort Funcionamento

2.2. Estudo de Complexidade do Bubble Sort

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do array. No pior caso, são feitas n^2 operações

Melhor Caso: O(n^2)

Pior Caso: O(n^2)

Espaço: O(1)

2.3. Testes Realizados no Bubble Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	49	0	0.000000
100	Ordenado	4950	0	0.000000
1000	Ordenado	499500	0	0.000001
10000	Ordenado	49995000	0	0.000129
100000	Ordenado	704982704	0	0.012833
1000000	Ordenado	1783293644	0	1.998644
10	Inverso	45	135	0.000000
100	Inverso	4950	14844	0.000000
1000	Inverso	499500	1498491	0.000016
10000	Inverso	49995000	149984142	0.000391
100000	Inverso	704982704	2114940081	0.047544
1000000	Inverso	1783293644	1051896305	4.897920
10	Aleatório	45	84	0.000000
100	Aleatório	4950	6282	0.000000
1000	Aleatório	499500	695625	0.000000
10000	Aleatório	49995000	67698672	0.000406
100000	Aleatório	704982704	806314449	0.028373
1000000	Aleatório	1783293664	801896196	2.920231
10	Quase Ordenado	45	0	0.000000
100	Quase Ordenado	4950	63	0.000000
1000	Quase Ordenado	499500	13845	0.000000
10000	Quase Ordenado	49995000	2302368	0.000203
100000	Quase Ordenado	704982704	62664396	0.021435
1000000	Quase Ordenado	1783293664	432911478	2.944792

3. Bubble Sort (Optimizado)

É possível optimizar o Bubble Sort para se obter um melhor caso de complexidade O(n), já o pior caso se mantem O(n^2).

3.1. Implementação

Uma maneira de se optimizar o Bubble Sort é adicionando uma variável para receber 0 ou 1, onde 1 significa que foi realizado uma troca e 0 que não foi realizado nenhuma operação durante o loop indicando que o vetor está ordenado, assim evita percorrer todo o vetor com ele já ordenado.

3.2. Testes Realizados no Bubble Sort(Optimizado)

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	19	30	0.000000
100	Ordenado	199	300	0.000000
1000	Ordenado	1999	3000	0.000000
10000	Ordenado	19999	30000	0.000000
100000	Ordenado	199999	300000	0.000000
1000000	Ordenado	1999999	3000000	0.000015
10	Inverso	54	162	0.000000
100	Inverso	5049	15141	0.000000
1000	Inverso	500499	1501488	0.000015
10000	Inverso	50004997	150014139	0.000796
100000	Inverso	705082703	2115240078	0.076213
1000000	Inverso	X	X	X
10	Aleatório	45	84	0.000000
100	Aleatório	4879	6609	0.000000
1000	Aleatório	499275	685257	0.000016
10000	Aleatório	50002444	6846324	0.000406
100000	Aleatório	156902781	80344553	0.012733
1000000	Aleatório	1876159650	813963171	0.087988
10	Quase Ordenado	19	30	0.000000
100	Quase Ordenado	2275	441	0.000000
1000	Quase Ordenado	471580	23022	0.000000
10000	Quase Ordenado	49592314	2170413	0.000203
100000	Quase Ordenado	643063481	63431067	0.009983
1000000	Quase Ordenado	1901315157	441416316	0.086626

4. Selection Sort

A ordenação por seleção é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos.[Wikipédia 2019f]

```c
void selection_Sort(float *vetor,int max){

  int min, aux;

  for (int i = 0; i < (max - 1); i++) {
       min = i;
    for (int j = i+1; j < max; j++) {

      if (vetor[j] < vetor[min]) {
          min = j;
      }
    }
      if (i != min) {
      aux = vetor[i];
      vetor[i] = vetor[min];
      vetor[min] = aux;
    }
  }
}
```

Figure 3. Selection Sort em C

4.1. Implementação

O selection sort compara a cada interação um elemento com os outros, visando encontrar o menor. Dessa forma, podemos entender que não existe um melhor caso mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois laços do algoritmo, o externo e o interno.

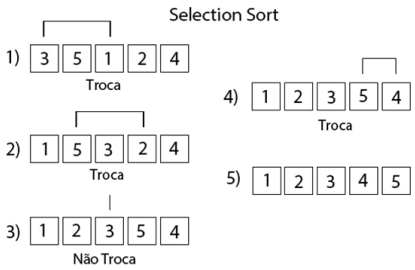

Figure 4. Selection Sort Funcionamento

4.2. Estudo de Complexidade do Selection Sort

Devido a execução dos dois laços no algoritmo a complexidade deste algoritmo será sempre O(n^2).

Melhor Caso: $O(n^2)$

Pior Caso: $O(n^2)$

Espaço: $O(1)$

4.3. Testes Realizados no Selection Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	45	27	0.000000
100	Ordenado	4950	297	0.000000
1000	Ordenado	499500	2997	0.000000
10000	Ordenado	49995000	29997	0.000187
100000	Ordenado	704982704	299997	0.017530
1000000	Ordenado	1783293644	2999997	1.740674
10	Inverso	45	27	0.000000
100	Inverso	4950	297	0.000000
1000	Inverso	499500	2997	0.000000
10000	Inverso	49995000	29997	0.00203
100000	Inverso	704982704	299997	0.017843
1000000	Inverso	1783293664	2999997	2.356542
10	Aleatório	45	27	0.000000
100	Aleatório	4950	297	0.000000
1000	Aleatório	499500	2997	0.000000
10000	Aleatório	49995000	29997	000.172
100000	Aleatório	704982704	299997	0.017577
1000000	Aleatório	1783293664	2999997	2.317490
10	Quase Ordenado	45	27	0.000000
100	Quase Ordenado	4950	297	0.000000
1000	Quase Ordenado	499500	2997	0.000000
10000	Quase Ordenado	49995000	29997	0.000172
100000	Quase Ordenado	704982704	299997	0.017499
1000000	Quase Ordenado	1783293664	2999997	2.330977

5. Selection Sort (Optimizado)

Não foram realizados testes em sua versão optimizada.

Pesquisa:

Sobre a implementação de uma verificação (min == i) para evitar a troca, no método Select Sort, não altera a complexidade do algoritmo devido a execução dos dois laços, permanecendo assim no melhor e pior caso O(n^2) mas melhora o seu tempo de execução.

Para torná-lo mais eficiente pois para fazer a troca antes verificamos se o Min é diferente do i, caso seja verdadeiro então realizamos a troca,uma vez que não era necessário fazer essas operações quando o Min for igual ao i. Assim fazemos menos atribuições e consequentemente diminuímos o tempo de execução do programa, o que ajuda na ordenação de grandes números.

6. Insertion Sort

Insertion Sort, ou ordenação por inserção, é o algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

```c
void insertion_Sort(float vetor[], int tam){

    int key, j, i;

    for (int i = 1; i < tam; i++){
        key = vetor[i];
        j = i - 1;

       while ((j >= 0) && (vetor[j] > key))
        {
            vetor[j + 1] = vetor[j];
            j--;
        }

        vetor[j + 1] = key;
    }
}
```

Figure 5. Insertion Sort em C

6.1. Implementação

O Insertion Sort inicia a ordenação dividindo o vetor em duas partições: uma ordenada à esquerda e outra não ordenada à direita. Inicialmente, a partição esquerda só terá um elemento (é o caso trivial da ordenação).[Wikipédia 2019c]

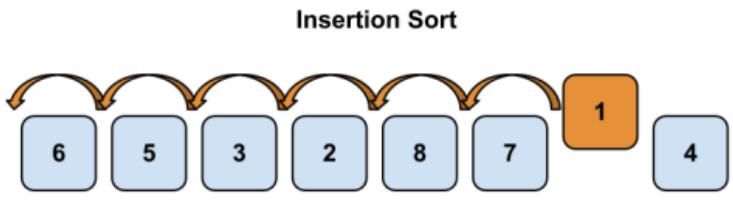

Figure 6. Insertion Sort Funcionamento

6.2. Estudo de complexidade do Insertion Sort

O pior caso do Insertion Sort ocorre quando os elementos do vetor estão em ordem decrescente, pois a condição ($A[j] > elemento$) sempre será verdadeira. Logo, o laço interno realizará a quantidade máxima de iterações. Nesse caso, o Insertion Sort terá complexidade no tempo de O(n^2).

O melhor caso ocorre quando o vetor está ordenado, pois, ao contrário do pior caso, a condição ($A[j] > Key$) sempre será falsa. Logo, o código do laço interno nunca será executado. Assim, teremos apenas as (n-1) interações do laço externo, portanto a complexidade no tempo é O(n).

Melhor Caso: $O(n)$

Pior Caso: $O(n^2)$

Espaço: $O(1)$

6.3. Testes Realizados no Insertion Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	0	0	0.000000
100	Ordenado	0	0	0.000000
1000	Ordenado	0	0	0.000000
10000	Ordenado	0	0	0.000000
100000	Ordenado	0	0	0.000000
1000000	Ordenado	0	0	0.000016
10	Inverso	0	45	0.000000
100	Inverso	0	4948	0.000000
1000	Inverso	0	499497	0.000004
10000	Inverso	0	49994714	0.000162
100000	Inverso	0	704980027	0.021763
1000000	Inverso	0	X	X
10	Aleatório	0	22	0.000000
100	Aleatório	0	2134	0.000000
1000	Aleatório	0	233375	0.000002
10000	Aleatório	0	22646302	0.000086
100000	Aleatório	0	26869264	0.001340
1000000	Aleatório	0	26869264	0.000893
10	Quase Ordenado	0	0	0.000000
100	Quase Ordenado	0	54	0.000000
1000	Quase Ordenado	0	8772	0.000000
10000	Quase Ordenado	0	645466	0.000002
100000	Quase Ordenado	0	21377076	0.000066
1000000	Quase Ordenado	0	144962499	0.000457

7. Insertion Sort (Optimização)

Utiliza-se o Sentinela, um registro na posição 0 do vetor que quando j=0 indicará o fim do loop, indicando que o vetor estará ordenado.

```c
void insertionSortO(TArray* pA,long* att,long* comp){

    int i, j;

    TItem temp, *arraySentinel = malloc((pA->n + 1) * sizeof(TItem));

    for(i = 0; i < pA->n; i++){
        arraySentinel[i+1] = pA->item[i];
        (*att)++;
    }
    arraySentinel[0].key = INT_MIN;

    for (i = 1; i < (pA->n + 1); i++) {
            temp = arraySentinel[i];
            j = i;
            (*att) += 2;
            while (arraySentinel[j - 1].key > temp.key) {
                arraySentinel[j] = arraySentinel[j - 1];
                j--;
                (*att)++;
                (*comp)++;
            }
            arraySentinel[j] = temp;
    }

    for (int i = 1; i < (pA->n + 1); i++){
        pA->item[i-1] = arraySentinel[i];
        (*att)++;
    }
}
```

Figure 7. Insertion Sort (Optimizado) em C

7.1. Testes Realizados com o Insertion Sort (Optimizado)

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	0	40	0.000000
100	Ordenado	0	400	0.000000
1000	Ordenado	0	4000	0.000000
10000	Ordenado	0	40000	0.000000
100000	Ordenado	0	400000	0.000002
1000000	Ordenado	0	4000000	0.000032
10	Inverso	45	85	0.000000
100	Inverso	4948	5348	0.000000
1000	Inverso	499497	503497	0.000004
10000	Inverso	4994714	50034714	0.000243
100000	Inverso	X	X	X
1000000	Inverso	1783287887	1787287887	2.149320
10	Aleatório	24	64	0.000000
100	Aleatório	2102	2502	0.000000
1000	Aleatório	228836	232836	0.000002
10000	Aleatório	22715469	22755469	0.000167
100000	Aleatório	267836858	268236858	0.001052
1000000	Aleatório	268554101	272554101	0.000884
10	Quase Ordenado	0	40	0.000000
100	Quase Ordenado	75	475	0.000000
1000	Quase Ordenado	6032	10032	0.000000
10000	Quase Ordenado	719939	759939	0.000005
100000	Quase Ordenado	20739710	21139710	0.000132
1000000	Quase Ordenado	145468074	149468074	0.000499

8. Shell Sort

Shell sort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o array a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles.[Wikipédia 2019g]

```
void shell_sort(TArray* pA, int tamanho,long* att, long*comp)
{
    int i , j , value;
    int gap = 1;

    do {
         gap = 3*gap+1;
       } while(gap < tamanho);

    do {
        gap /= 3;

    for(i = gap; i < tamanho; i++) {

        value = pA->item[i].key;
        j = i - gap;
        (*att)+=1;
        (*comp)+=2;

    while (j >= 0 && value < pA->item[j].key) {

            (*att)+=1;
        pA->item[j + gap] = pA->item[j];
        j -= gap;
    }

    pA->item[j + gap].key = value;
    }
    }while(gap > 1);
}
```

Figure 8. Shell Sort em C

8.1. Implementação

Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção.

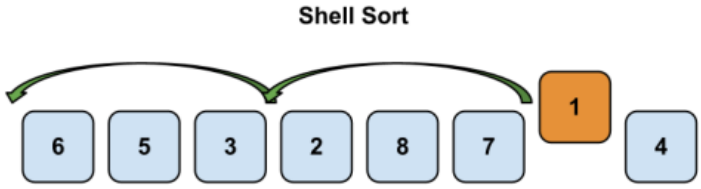

Figure 9. Shell Sort Funcionameto

8.2. Estudo de Complexidade do Shell Sort

O algoritmo não possui fórmula fechada para sua função de complexidade.

A complexidade do algoritmo ainda não é conhecida, sua análise contém alguns problemas matemáticos muito difíceis, Exemplo: escolher a sequencia de incrementos. O que se sabe é que cada incremento não deve ser múltiplo do anterior.

8.3. Testes Realizados no Shell Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	30	12	0.000000
100	Ordenado	684	342	0.000000
1000	Ordenado	10914	5457	0.000000
10000	Ordenado	150486	75243	0.000000
100000	Ordenado	1934292	967146	0.000005
1000000	Ordenado	23608530	11804265	0.000094
10	Inverso	30	28	0.000000
100	Inverso	684	570	0.000000
1000	Inverso	10914	9377	0.000000
10000	Inverso	150486	128789	0.000000
100000	Inverso	1934292	1585277	0.000006
1000000	Inverso	23608530	18046294	0.000083
10	Aleatório	30	29	0.000000
100	Aleatório	684	745	0.000000
1000	Aleatório	10914	13946	0.000000
10000	Aleatório	150486	237018	0.000001
100000	Aleatório	1934292	1668408	0.000008
1000000	Aleatório	23608530	12579833	0.000066
10	Quase Ordenado	30	15	0.000000
100	Quase Ordenado	684	411	0.000000
1000	Quase Ordenado	10914	7584	0.000000
10000	Quase Ordenado	150486	139538	0.000001
100000	Quase Ordenado	1934292	1377344	0.000008
1000000	Quase Ordenado	23608530	12474459	0.000058

9. Quick Sort

O algoritmo quicksort é um método de ordenação muito rápido e eficiente. O quicksort adota a estratégia de divisão e conquista.[Wikipédia 2019e]

```c
void quick_Sort(TArray* vetor,int esquerda,int direita,long* att,long* comp){

    int i, j, pivo;
    TItem aux;
    i = esquerda;
    j = direita-1;
    pivo = vetor->item[(esquerda + direita) / 2].key;
    while(i <= j)
    {
        while(vetor->item[i].key < pivo && i < direita)
        {
            i++;
        }
        while(vetor->item[j].key > pivo && j > esquerda)
        {
            j--;
        }
        (*comp)++;
        if(i <= j)
        {
            (*att)+=3;
            aux = vetor->item[i];
            vetor->item[i] = vetor->item[j];
            vetor->item[j] = aux;
            i++;
            j--;
        }
    }
    (*comp)++;
    if(j > esquerda){

        quick_Sort(vetor, esquerda, j+1,att,comp);
    }
    (*comp)++;
    if(i < direita){

        quick_Sort(vetor, i, direita,att,comp);
    }
}
```

Figure 10. Quick Sort em C

9.1. Implementação

A estratégia consiste em rearranjar as chaves de modo que as chaves ”menores” precedam as chaves ”maiores”, o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

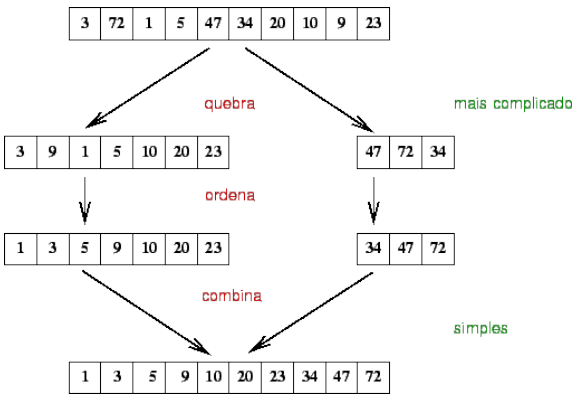

Figure 11. Quick Sort em C

9.2. Estudo de Complexidade do Quick Sort

O melhor caso é quando o pivô sempre fica no meio. Nesse caso, podemos considerar que as duas partes têm a metade de elementos do problema original.

Melhor Caso: $O(n.logn)$

No pior caso, o pivô está sempre na primeira ou última posição.

Pior Caso: $O(n^2)$

Espaço: $O(n)$

9.3. Testes Realizados no Quick Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	21	21	0.000000
100	Ordenado	210	210	0.000000
1000	Ordenado	2265	2265	0.000000
10000	Ordenado	24945	24945	0.000001
100000	Ordenado	210762	210762	0.000005
1000000	Ordenado	2233467	2233467	0.000094
10	Inverso	25	33	0.000000
100	Inverso	262	360	0.000000
1000	Inverso	2740	3738	0.000000
10000	Inverso	29395	39393	0.000001
100000	Inverso	257719	357717	0.000007
1000000	Inverso	2725030	3725028	0.000052
10	Aleatório	33	36	0.000000
100	Aleatório	449	594	0.000000
1000	Aleatório	4997	7719	0.000000
10000	Aleatório	53710	97047	0.000002
100000	Aleatório	364535	544314	0.000015
1000000	Aleatório	2383005	2558574	0.000082
10	Quase Ordenado	21	21	0.000000
100	Quase Ordenado	211	213	0.000000
1000	Quase Ordenado	2281	2313	0.000000
10000	Quase Ordenado	25304	26016	0.000001
100000	Quase Ordenado	217573	227391	0.000004
1000000	Quase Ordenado	2295015	2381052	0.000087

10. Heap Sort

O algoritmo heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção.[Wikipédia 2019b]

```
void criarHeap(TArray* pA, int inicio, int fim,long* att,long* comp){

    TItem aux = pA->item[inicio];
    int filho = (inicio * 2)+1;

    while(filho<=fim){
            (*comp)+=2;
    if(filho<fim && (filho+1)<fim){
        if(pA->item[filho].key <pA->item[filho+1].key){

            filho++;
        }
    }
    if(aux.key<pA->item[filho].key){
            (*att)+=1;
        pA->item[inicio]=pA->item[filho];
        inicio=filho;
        filho=(2*inicio)+1;
    }
    else{

        filho=fim+1;
    }
}
    pA->item[inicio] = aux;
}
void heap_sort(TArray* pA, int tamanho,long* att,long* comp){

    int i;
    TItem aux;
    for(i=(tamanho-1)/2;i>=0;i--){

        criarHeap(pA,i,tamanho-1,att,comp);
    }
    for(i=tamanho-1;i>=1;i--){

            (*att)+=3;
        aux = pA->item[0];
        pA->item[0]=pA->item[i];
        pA->item[i]=aux;
        criarHeap(pA,0,i-1,att,comp);
    }
}
```

Figure 12. Heap Sort em C

10.1. Implementação

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap. A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor.

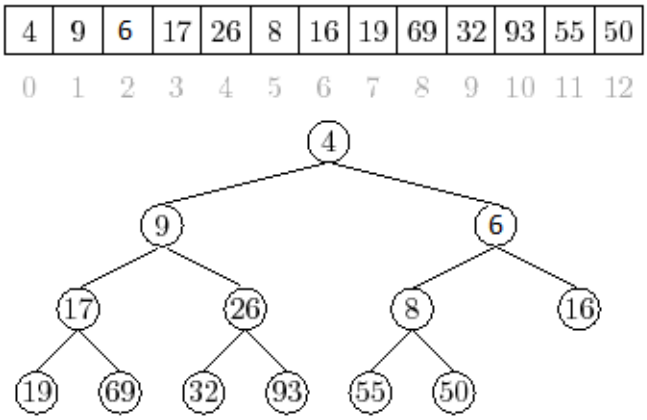

Figure 13. Heap Sort Funcionamento

10.2. Estudo de Complexidade do Heap Sort

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio.

O heapsort trabalha no lugar e o tempo de execução em pior cenário para ordenar n elementos é de O (n lg n). Lê-se logaritmo (ou log) de ”n” na base 2. Para valores de n, razoavelmente grandes, o termo log n é quase constante, de modo que o tempo de ordenação é quase linear com o número de itens a ordenar.

Melhor Caso: $O(n.logn)$.

Pior Caso: $O(n.longn)$.

Espaço: Total = $O(n)$, auxiliar $O(1)$.

O heapsort não é um algoritmo de ordenação estável. Porém, é possível adaptar a estrutura a ser ordenada de forma a tornar a ordenação estável, cada elemento da estrutura adaptada deve ficar no formato de um par (elemento original, índice original). Assim, caso dois elementos sejam iguais, o desempate ocorrerá pelo índice na estrutura original.

10.3. Testes Realizados no Heap Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	44	45	0.000000
100	Ordenado	1094	837	0.000000
1000	Ordenado	17632	11696	0.000000
10000	Ordenado	244516	151522	0.000003
100000	Ordenado	3113240	1851208	0.000030
1000000	Ordenado	37727904	21788227	0.000234
10	Inverso	38	39	0.000000
100	Inverso	954	710	0.000000
1000	Inverso	15994	10320	0.000000
10000	Inverso	226744	136692	0.000003
100000	Inverso	2926814	1697551	0.000023
1000000	Inverso	36003078	20332992	0.000193
10	Aleatório	46	46	0.000000
100	Aleatório	1052	794	0.000000
1000	Aleatório	16978	11171	0.000000
10000	Aleatório	236370	145107	0.000004
100000	Aleatório	3092276	1834504	0.000030
1000000	Aleatório	37708462	21773287	0.000255
10	Quase Ordenado	44	45	0.000000
100	Quase Ordenado	1086	833	0.000000
1000	Quase Ordenado	17580	11671	0.000000
10000	Quase Ordenado	244320	151319	0.000003
100000	Quase Ordenado	3112108	1850115	0.000030
1000000	Quase Ordenado	37718638	21780868	0.000184

11. Merge Sort

O merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.[Wikipédia 2019d]

```
void merge(TArray* vetor, int comeco, int meio, int fim,long* att,long* comp)

    int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;
    int *vetAux;
    vetAux = (int*)malloc(tam * sizeof(int));

    while(com1 <= meio && com2 <= fim){
        (*comp)++;
        if(vetor->item[com1].key < vetor->item[com2].key) {
                (*att)+=1;
            vetAux[comAux] = vetor->item[com1].key;
            com1++;
        } else {
                (*att)+=1;
            vetAux[comAux] = vetor->item[com2].key;
            com2++;
        }
        comAux++;
    }
    (*comp)++;
    while(com1 <= meio){

            (*att)+=1;
        vetAux[comAux] = vetor->item[com1].key;
        comAux++;
        com1++;
    }
    (*comp)++;
    while(com2 <= fim) {

             (*att)+=1;
        vetAux[comAux] = vetor->item[com2].key;
        comAux++;
        com2++;
    }

    for(comAux = comeco; comAux <= fim; comAux++){

             (*att)+=1;
        vetor->item[comAux].key = vetAux[comAux-comeco];
    }

    free(vetAux);
}
```

Figure 14. Merge em C

```
void merge_sort(TArray* vetor, int comeco, int fim,long* att,long* comp){

    if (comeco < fim) {
        int meio = (fim+comeco)/2;

       merge_sort(vetor, comeco, meio,att,comp);
       merge_sort(vetor, meio+1, fim,att,comp);
       merge(vetor, comeco, meio, fim,att,comp);
    }
}
```

Figure 15. Merge Sort em C

11.1. Implementação

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).

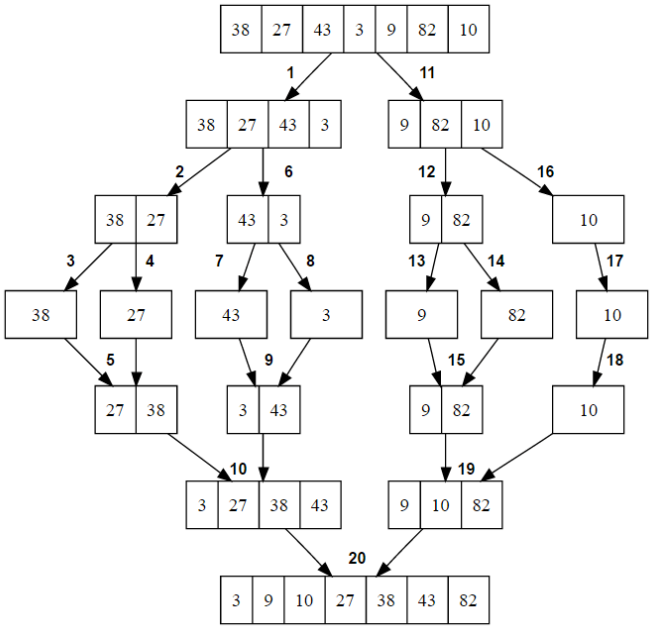

Figure 16. Merge Sort Funcionamento

11.2. Estudo de Complexidade do Merge Sort

Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

É possível implementar o merge sort utilizando somente um vetor auxiliar ao longo de toda a execução, tornando assim a complexidade de espaço adicional igual a $(nlong(n))$.

É um algoritmo estável na maioria das implementações, em que elas podem ser iterativas ou recursivas.

É possível também implementar o algoritmo com espaço adicional

Melhor caso: $O(nlog(n))$.

Pior caso: $O(nlog(n))$, variante natural $O(n)$.

Espaço: $O(nlog(n))$.

11.3. Testes Realizados no Merge Sort

Array de Ordem	Estado do Array	Comparações	Movimentações	Tempo(s)
10	Ordenado	37	68	0.000000
100	Ordenado	554	1344	0.000001
1000	Ordenado	7051	19952	0.000001
10000	Ordenado	89239	267232	0.000005
100000	Ordenado	1055192	3337856	0.000032
1000000	Ordenado	12071033	39902848	0.000265
10	Inverso	33	68	0.000000
100	Inverso	514	1344	0.000001
1000	Inverso	6930	19952	0.000001
10000	Inverso	84606	267232	0.000005
100000	Inverso	1015022	3337856	0.000031
1000000	Inverso	11884990	39902848	0.000239
10	Aleatório	42	68	0.000001
100	Aleatório	743	1344	0.000000
1000	Aleatório	10652	19952	0.000001
10000	Aleatório	140427	267232	0.000007
100000	Aleatório	1245559	3337856	0.000044
1000000	Aleatório	12268913	39902848	0.000265
10	Quase Ordenado	37	68	0.000001
100	Quase Ordenado	561	1344	0.000000
1000	Quase Ordenado	8756	19952	0.000000
10000	Quase Ordenado	121042	267232	0.000007
100000	Quase Ordenado	1209129	3337856	0.000040
1000000	Quase Ordenado	12264517	39902848	0.000312

12. Análise Algoritmos com caso: $O(n^2)$ versus $O(nlog(n))$

Com análises feitas nos graficos a seguir é possivel obter o Melhor caso e o Pior caso de cada algoritmo apresentado, considerando as entradas (Array) contendo $(1.000.000)$ de elementos.

Algoritmo	Melhor Caso	Pior Caso
Bubble Sort	Ordenado	Inverso
Selection Sort	Ordenado	Inverso - Aleatório - Quase Ordenado
Insertion Sort	Ordenado	Inverso
Shell Sort	Quase ordenado	Ordenado
Quick Sort	Inverso	Ordenado
Heap Sort	Quase Ordenado	Aleatório
Merge Sort	Inverso	Quase ordenado

Para arrays que possuem poucos elementos(10,100,1.000) a utilização de algoritmos com casos $O(n^2)$ é mais eficiente, devido ao custo de memória não ser tão elevado, já para arrays com muitos elementos (10.000,100.000,1.000.000) é recomentado a utilização de algoritmos com caso $O(nlog(n))$, devido a velocidade de resposta do algoritmo para fornecer o resultado desejado.

Com Arrays de (1.000.000) o tempo de execução de algoritmos com caso $O(n^2)$ chega a durar horas, já o mesmo array em algoritmos $O(nlog(n))$ é processado em poucos segundos.

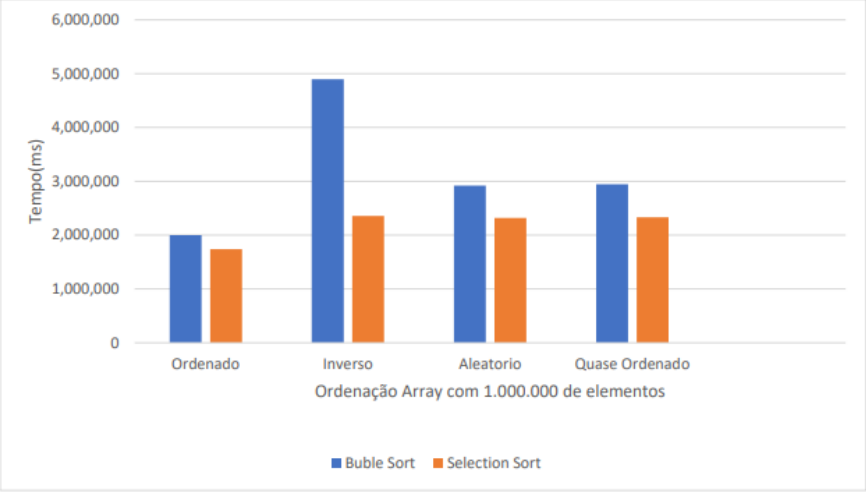

Figure 17. Algoritmos $O(n^2)$

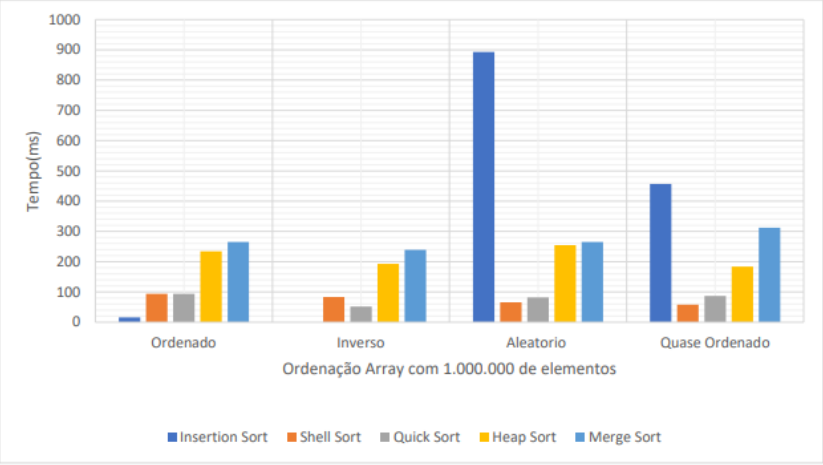

Figure 18. Algoritmos $O(n)$ e $O(nlog(n))$

13. Conclusão

Cada algoritmo apresentado pode ser usado em diversas aplicações, tendo em vista a necessidade de ordenação basta analisarmos qual é a proporção dos elementos a serem organizados, assim podemos selecionar o melhor algoritmo para suprir nossa necessidade.

13.1. Dificuldades encontradas no trabalho

Conteúdo sobre as optimizaçoes dos algoritmos de caso $O(n^2)$, Selection Sort, Insertion Sort, Bubble Sort.

Construção dos gráficos para análise devido a quantidade de conteúdo a ser analisado.

References

Wikipédia (2019a). Bubble sort — wikipédia, a enciclopédia livre. [Online; accessed 21-abril-2019].

Wikipédia (2019b). Heapsort — wikipédia, a enciclopédia livre. [Online; accessed 20-abril-2019].

Wikipédia (2019c). Insertion sort — wikipédia, a enciclopédia livre. [Online; accessed 5-maio-2019].

Wikipédia (2019d). Merge sort — wikipédia, a enciclopédia livre. [Online; accessed 23-março-2019].

Wikipédia (2019e). Quicksort — wikipédia, a enciclopédia livre. [Online; accessed 13-maio-2019].

Wikipédia (2019f). Selection sort — wikipédia, a enciclopédia livre. [Online; accessed 29-maio-2019].

Wikipédia (2019g). Shell sort — wikipédia, a enciclopédia livre. [Online; accessed 22-maio-2019].