Análise de complexidade de algoritmo de ordenação

1- Buble Sort: É um dos mais simples algoritmos de ordenação, a ideia dele é percorrer um vector várias vezes, e a cada passagem ele muda o maior elemento para o final do vector.
No melhor caso ele não executaria nenhuma vez, ou seja, o vector já estaria ordenado, não haveria nenhuma troca, e faríamos apenas um loop, então teremos O(n).
No pior caso a ordenação do vetor estaria em ordem reversa, ou seja, seria necessário
executa n vezes, isto é, o número total de elementos no vector, então teremos $O(n^2)$.

```
void Ordenacao::bubbleSort()
{
    int* copia = (int*) malloc(this->tamanho * sizeof(int));
    for (int i=0; i < this->tamanho; i++){
        copia[i] = this->vetor[i];
    }

    int k, j, aux;
    for(k = 0; k < this->tamanho; k++)
    {
        for(j = 0; j < this->tamanho - 1; j++)
        {
            if(copia[j] > copia[j + 1])
            {
                aux = copia[j];
                copia[j] = copia[j + 1];
                copia[j + 1] = aux;
            }
        }
    }
}
```

2- Insert Sort: É uma estrutura de ordenação indicada para organizar um vetor de acordo com a inserção de elementos. Ele pega um vetor e a cada inserção ele verifica se o elemento inserido e maior do que o elemento que ele esta percorrendo, caso seja maior ele continua percorrendo o vetor até que o elemento que está sendo inserido seja o menor elemento.
No melhor caso o elemento que está sendo inserido estará na primeira posição do vetor, então teremos O(n).
No piro caso o elemento que será inserido estará no final do vetor, ou seja, será necessário percorrer todo o vetor para inserir o elemento, então teremos O(n).

```
void Ordenacao::insertionSort() {

    int aux;
    int* copia = (int*) malloc(this->tamanho * sizeof(int));

    for (int i=0; i < this->tamanho; i++){
        copia[i] = this->vetor[i];
    }

    for (int i = 1; i < this->tamanho; i++) {
        int escolhido = copia[i];
        int j = i - 1;

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

        copia[j + 1] = escolhido;
    }
}
```

3- Selection Sort: A ordenação por seleção visa selecionar o menor elemento do vetor e colocar ele na primeira no primeiro índice, para isto é criado dois loops um para armazenar o índice do vetor e outro para percorrer o vetor e localizar o menor elemento, a cada ciclo é localizado o menor elemento e este e colocado na primeira posição, com isso o primeiro loop soma mais 1, e agora começa a verificação através do segundo índice, pois o primeiro índice já contém o menor elemento, e assim sucessivamente até o último elemento. Pelo fator do algoritmo precisar percorrer todo o vetor para comparar o menor elemento com os outros, ele sempre vai precisar percorrer todo o vetor para localizar a posição desejada, com isto ele não possui um melhor caso, ou seja, ele sempre vai ser O(n).

```
void Ordenacao::selectSort(){
    int aux;
    int* copia = (int*) malloc(this->tamanho * sizeof(int));

    for (int i=0; i < this->tamanho; i++){
        copia[i] = this->vetor[i];
    }

    for (int indice = 0; indice < tamanho; ++indice) {

        int indiceMenor = indice;
        for (int indiceSeguinte = indice+1; indiceSeguinte < tamanho; ++indiceSeguinte) {
            if (copia[indiceSeguinte] < copia[indiceMenor]) {
                indiceMenor = indiceSeguinte;
            }
        }
        int aux = copia[indice];
        copia[indice] = copia[indiceMenor];
        copia[indiceMenor] = aux;
    }
}
```

4- O Quicksort é o algoritmo mais eficiente na ordenação por comparação. Nele se escolhe um elemento chamado de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada.
A complexidade é aproximadamente O(nlog(n)) quando a seleção do pivô divide o array original em dois sub arrays de tamanhos quase iguais.
Por outro lado, se o algoritmo, que seleciona o elemento pivô dos arrays de entrada, produz consistentemente 2 sub-arrays com uma grande diferença em termos de tamanho, o algoritmo de ordenação rápida pode atingir a complexidade temporal de pior caso de O(n^2).

```
void Ordenacao::quicksort(int* copia, int inicio, int fim){

    int i, j, pivo, aux;
    i = inicio;
    j = fim-1;
    pivo = copia[(inicio + fim) / 2];
    while(i <= j)
    {
        while(copia[i] < pivo && i < fim)
        {
            i++;
        }
        while(copia[j] > pivo && j > inicio)
        {
            j--;
        }
        if(i <= j)
        {
            aux = copia[i];
            copia[i] = copia[j];
            copia[j] = aux;
            i++;
            j--;
        }
    }
    if(j > inicio)
        quicksort(copia, inicio, j+1);
    if(i < fim)
        quicksort(copia, i, fim);
}
```

5- O Shell Sort se baseia em uma variável chamada de incremento de sequência, ou incremento de shell, que é dado por h e ao decorrer da execução do algoritmo, é decrementada até 1.
Utilizando o incremento de shell, o algoritmo compara elementos distantes em um vetor, em vez de comparar os adjacentes.

No algoritmo, a ordenação é realizada em vários passos, usando uma sequência de valores do incremento de shell <h1, h2, h3…hN> onde começando por hN selecionamos apenas os valores que estão hN elementos distantes um do outro, então ordenamos esses elementos com algum algoritmo de ordenação simples como bolha, seleção ou inserção. Deste modo, apenas os elementos selecionados serão ordenados, os outros são todos ignorados.

```
void Ordenacao::shellSort() {

    int aux;
    int* copia = (int*) malloc(this->tamanho * sizeof(int));

    for (int i=0; i < this->tamanho; i++){
        copia[i] = this->vetor[i];
    }

    int i, j, value;
    int h = 1;
    while(h < this->tamanho) {
        h = 3*h+1;
    }
    while (h > 0) {
        for(i = h; i < this->tamanho; i++) {
            value = copia[i];
            j = i;
            while (j > h-1 && value <= copia[j - h]) {
                copia[j] = copia[j - h];
                j = j - h;
            }
            copia[j] = value;
        }
        h = h/3;
    }
}
```

6- Merge é a rotina que combina dois arrays ordenados em um outro também ordenado. Assim como o Quick Sort aplica várias vezes o particionamento para ordenar um array, o Merge Sort também aplica o Merge várias para ordenar um array. O objetivo é organizar os dados no array a ser ordenado de forma que uma parte dele esteja ordenada e outra também. Assim, no Merge Sort não fazemos o merge de dois arrays, mas fazemos o merge de duas partes ordenadas de um mesmo array.
Independente do caso (melhor, pior ou médio) o Merge Sort sempre será $n*\log n$. Isso ocorre porque a divisão do problema sempre gera dois sub-problemas com a metade do tamanho do problema original ($2*T(n/2)$).
O algoritmo de Merge é O(n).

```
void Ordenacao::merge(int* copia, int ini, int meio, int fim, int* vetor2) {
    int esq = ini;
    int dir = meio;
    for (int i = ini; i < fim; ++i) {
        if ((esq < meio) and ((dir >= fim) or (vetor[esq] < vetor[dir]))) {
            vetor2[i] = copia[esq];
            ++esq;
        }
        else {
            vetor2[i] = copia[dir];
            ++dir;
        }
    }
    for (int i = ini; i < fim; ++i) {
        copia[i] = vetor2[i];
    }
}

void Ordenacao::mergeSort(int* copia, int inicio, int fim, int* vetor2) {
    if ((fim - inicio) < 2) return;

    int meio = ((inicio + fim)/2);
    mergeSort(copia, inicio, meio, vetor2);
    mergeSort(copia, meio, fim, vetor2);
    merge(copia, inicio, meio, fim, vetor2);
}
```

7- Radix Sort: Tem como ideia é fazer a ordenação dígito a dígito começando do dígito menos significativo para o dígito mais significativo. A classificação Radix usa a classificação por contagem como uma sub-rotina para classificar.
Sejam d dígitos nos inteiros de entrada. Radix Sort leva tempo O(d*(n+b)) onde b é a base para representar números, por exemplo, para o sistema decimal, b é 10. Qual é o valor de d? Se k é o valor máximo possível, então d seria O(log b (k)). Portanto, a complexidade de tempo total é O((n+b) * log b (k)). O que parece mais do que a complexidade de tempo dos algoritmos de ordenação baseados em comparação para um grande k. Vamos primeiro limitar k. Seja k <= n c onde c é uma constante. Nesse caso, a complexidade se torna O(nLog b (n)). Mas ainda não supera os algoritmos de classificação baseados em comparação.

```
void Ordenacao::radixSort() {
    int* copia = (int*) malloc(this->tamanho * sizeof(int));

    for (int i=0; i < this->tamanho; i++){
        copia[i] = this->vetor[i];
    }

    int i;
    int *b;
    int maior = 0;
    int exp = 1;

    b = (int *)calloc(this->tamanho, sizeof(int));

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

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

Bibiografia.

https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o

https://www.geeksforgeeks.org/radix-sort/