COMPLEXIDADE DOS ALGORITMOS E ANÁLISE DOS MÉTODOS DE ORDENAÇÃO

Gabriel Tonhatti Cardoso[1]

Resumo

A Teoria da Complexidade dos Algoritmos é uma medida computacional em que são medidos recursos necessários para especificar o objeto, geralmente computando requisitos de tempo ou espaço de memória e é também conhecida como a complexidade algorítmica. Assim, a teoria é embasada no conceito de delimitar um certo conjunto de problemas, cujo consumo de recursos pode ser mensurado. Então, a partir dos problemas de condição são considerados os problemas matemáticos. Isso requer uma resposta que descreva a solução algorítmica. A partir das relações funcionais estabelecidas, a resolução consiste na exigência de determinar o máximo ou mínimo de uma dada função de custo sobre todas as soluções possíveis para o problema.

Palavras-chave: Análise de complexidade dos algoritmos, métodos de ordenação.

Abstract

Algorithm Complexity Theory is a computational measure that measures the necessary resources to specify an object by usually calculating time or memory space requirements, which is also known as algorithmic complexity. Therefore, the theory is based on the concept of delimiting a certain set of problems whose resource consumption can be measured. Then, starting from the conditional problems the mathematical problems are considered, which requires an answer that describes an algorithmic solution. Based on the established functional relationships, the resolution consists of the demand to determine the maximum or minimum of a given cost function of all possible solutions to the problem.

Keywords: *Algorithm complexity analysis, ordering methods.*

1 Introdução

Existe vários métodos de ordenação que são usados com as linguagens de programação para se ordenar dados de um Array ou lista. Com isso, pensando em entender a complexidade dos algoritmos, existem diversos algoritmos de ordenação, mas nesse trabalho será analisado a complexidade dos métodos de ordenação

[1] Graduando em Análise e Desenvolvimento de Sistemas pela Fatec Dr Thomaz Novelino – Franca/SP. Endereço eletrônico: gabriel.cardoso30@fatec.sp.gov.br.

Bubble Sort, Selection Sort, Merge Sort e Quick Sort, descrevendo sobre a complexidade que cada um dos algoritmos.

1.1 Algoritmos

Algoritmos são sequências de passos para se realizar uma tarefa, o exemplo mais famoso é uma receita de bolo, onde se tem o passo a passo de como preparar e fazer um bolo, da mesma forma na ciência da computação tem a sequência de passos que é escrito no código para se definir alguma ação a ser feita por um programa, podendo receber um valor de entrada e retornar um valor de saída.

Exemplos de problemas que envolvem algoritmos no dia a dia:

- Trocar uma lâmpada.
- Preparar um bolo.
- Organizar a lista de contatos por ordem alfabética.
- Calcular a rota mais curta entre duas ruas.
- Calcular a rota mais curta entre duas ruas

De forma geral, algoritmos são usados como ferramentas para decifrar um problema, as suas características são: finitas, bem definidas e efetivas.

As principais técnicas são:

- Algoritmo guloso
- Programação Dinâmica
- Dividir e Conquistar
- Busca e Ordenação
- Backtracking

Também são utilizadas as **Estrutura de Dados** juntamente com essas técnicas para ajudar numa melhor eficácia dos algoritmos. Como por exemplo: Grafos, Árvores, Heaps, Tabelas Hash, Pilhas e Filas.

1.2 Eficiência dos Algoritmos

Existe vários métodos para se analisar a performance e eficácia de algum programa, os famosos “Profilers”. Contudo, embora tenha uma boa eficácia, eles não são tão conveniente para analisar a complexidade de algoritmos. A complexidade dos

algoritmos avaliam um algoritmo em “nível de idealização/definição”, ou seja, desconsiderando qualquer implementação de linguagens específicas ou hardware. Comparar o tempo que um algoritmo leva para executar determinada tarefa em milissegundos em relação a outro, não é características para dizer que um algoritmo é melhor do que o outro.

Pode-se dizer que o melhor algoritmo para decifrar um problema é aquele que tem a menor complexidade de tempo e espaço. Ou seja, é aquele algoritmo em que, de acordo com que a entrada cresce visando o infinito, é aquele que mostra o menor tempo de variação e a menor memória utilizada para terminar.

Um algoritmo é capaz de ser melhor do que outro quando processa pouco dados, mas é capaz de ser muito pior de acordo com que os dados crescem.

A **Análise de complexidade** nos permite medir o quão rápido um programa executa seus cálculos.

1.3 Comportamento Assintótico

Seria um trabalho muito cansativo ficar contando a quantidade de informação para cada trecho de código que fosse escrito. Além disso, a quantidade de informações varia muito de linguagem, compilador e até mesmo o hardware da máquina que está sendo utilizada.

Na **Análise de Complexidade** podemos apenas nos interessar pelo termo que mais cresce de acordo com a entrada. Para atingirmos esse termo, podemos remover todas as constantes e manter apenas o termo que mais cresce.

No método **f(n) = 6n + 4**, evidentemente, 4 continua com o mesmo valor independente da entrada, mas **6n** aumenta cada vez. Retirando ele, encontramo-nos com o método **f(n) = 6n**.

Visto que 6 é uma constante, conseguimos retirá-lo e obtermos o método **f(n) = n**. Dessa forma, facilita demais a análise da complexidade do algoritmo.

Comportamento assintótico das seguintes funções:

- *f(n) = 5n + 12 nos retorna f(n) = n*

  Pelo mesmo fato do exemplo anterior.
- *f(n) = 915 nos retorna f(n) = 1*

  *Estamos retirando o multiplicador **915** * 1*
- *f(n) = n^2 + 2n + 300 nos retorna f(n) = n^2*

  Aqui, o termo n^2 aumenta mais rápido do que 2n
- *f(n) = n^2 + 2000n + 5929 nos retorna f(n) = n^2*

  Mesmo que o fator antes de n é bem grande, podemos encontrar um valor para n onde n^2 se torna maior que 2000n.

1.4 Complexidade dos Algoritmos e Notação Big-O

Para apresentar o comportamento assintótico de algum algoritmo, foi adotada a **Notação Big-O** pelo cientistas da computação. Ela é usada para demarcar assintoticamente o aumento(tempo ou espaço) superior do algoritmo.

Usando o algoritmo para encontrar o maior elemento como forma de demonstração, somos capazes de encontrar caso de entrada que fará com que ele execute um número menor de cálculos. Não será para todo caso de entrada que seu método para a quantidade de instruções seja **f(n)**.

Usando a notação Big-O, conseguimos dizer que a complexidade do algoritmo é **"Big-O de O(n)**", ou seja, no pior caso cresce em ordem de **n.**

Em algoritmos simples, é fácil identificar a complexidade do mesmo. Normalmente, se o algoritmo tem apenas 1 laço de repeitção, sua complexiade é **O(n)**, se possui 2 laços de repetição encadeados **O(n^2)** e se não possuir nenhum laço **O(1)**.

1.5 Complexidade de espaço

Toda os estudos feitos até o momento foram em função da quantidade de operações que os algoritmos pedem, que é semelhante à **Complexidade de Tempo**.

A complexidade de espaço de um algoritmo não é muito distinto da complexidade de tempo em questão de análise, que também é utilizado a notação **Big-O**.

Para conseguir analisar a complexidade de espaço de algum algoritmo, deve-se identificar a quantidade de memória que o algoritmo precisa alocar para resolver o problema no pior dos casos.

2 Métodos de Ordenação

Um método de ordenação é um algoritmo manipula os dados, para que coloque os elementos de uma dada sequência em uma certa ordem, para que possa facilitar a recuperação dos dados de uma lista.

2.1 Bubble Sort

Bubble Sort, também conhecido por “Ordenação por flutuação” ou “Ordenação por Bolha”, é um dos algoritmos de ordenação mais simples, que pode ser aplicado em Arrays ou lista. Basicamente o principio é percorrer o vetor(Array) varias vezes, e a cada passada fazer com que o maior elemento da sequência “flutue” para o topo.

No melhor caso do Bubble Sort, o algoritmo faz **n** operações essenciais, onde **n** representa a quantidade de elementos do vetor(Array). No pior caso do Bubble Sort, são feitas **n^2** operações.

Exemplo do algoritmo de ordenação BUBBLE SORT em JavaScript:

```
let pass, comps, trocas;

function bubbleSort(vetor) {

   pass = 0, comps = 0, trocas = 0;

   let trocou;

   do {
      pass++
      trocou = false;

      for (let i = 0; i < vetor.length - 1; i++) {
         comps++;
         if (vetor[i] > vetor[i + 1]) {
            [vetor[i], vetor[i + 1]] = [vetor[i + 1], vetor[i]];
            trocou = true;
```

```
            trocas++;
          }
        }

    } while (trocou)
}

let nums = [77, 44, 22, 33, 99, 55, 88, 0, 66, 11];
bubbleSort(nums);
console.log(nums);
```

2.2 Selection Sort

A ordenação por seleção ou Selecion Sort em inglês como é mais conhecido, baseia-se em pegar o menor elemento e colocar na primeira posição, selecionar o segundo menor valor para a segunda posição, e assim sucessivamente com os n - 1 elementos restante, até que percorra o vetor(Array) ou lista completamente. O Selection Sort tem complexidade **C(n) = O(n²)** em todos os casos(pior, médio e melhor caso), além de não ser um algoritmo estável.

O método Selection Sort é “in-place”, pois a ordenação é feita reordenando os elementos dentro do próprio vetor(Array), sem precisar de usar algum vetor(Array) auxiliar.

Vantagens de se usar o Selection Sort:

- Simples de ser implementado em relação a outros.
- Não necessita de um vetor(Array) auxiliar(in-place).
- Ocupa menos memória por não precisar de um auxiliar.
- É rápido em ordenação de vetores(Array) ou listas pequenas.

Desvantagens de se usar o Selection Sort:

- Não é estável.
- É muito lento para vetores(Arrays) ou listas muito grandes.
- Sempre faz **(n² - n) / 2** comparações.

Exemplo do algoritmo de ordenação SELECTION SORT em JavaScript:

```
let pass, comps, trocas;

function selectionSort(vetor) {

   pass = 0, comps = 0, trocas = 0;

   for (let posSel = 0; posSel < vetor.length - 1; posSel++) {

      pass++;

      let posMenor = posSel + 1;

      for (let i = posMenor + 1; i < vetor.length; i++) {
         if (vetor[posMenor] > vetor[i]) {
            posMenor = i;
         }
         comps++;
      }

      comps++;
      if (vetor[posSel] > vetor[posMenor]) {
         [vetor[posSel], vetor[posMenor]] = [vetor[posMenor], vetor[posSel]];
         trocas++;
      }
   }
}
let nums = [77, 44, 22, 33, 99, 55, 88, 0, 66, 11];
selectionSort(nums);
console.log(nums);
```

2.3 Merge Sort

Merge Sort ou “Ordenação por mistura”, foi criado pelo matemático John Von Neumann em 1945, é um algoritmo de ordenação por comparação que faz uso da estratégia “Dividir para conquistar”.

A ideia principal desse algoritmo é **Dividir**(o problema em problemas menores (subproblemas) e resolver esses problemas usando a recursividade) e **Conquistar**(Unir novamente todas as soluções dos “subproblemas” após todos terem sido resolvidos). O Merge Sort é estável, mas pelo fato de usar da recursividade ele acaba que consome bastante memória.

Complexidade do Merge Sort:

- Complexidade de tempo: **O($n \log_2 n$).**

- Complexidade de espaço: **O(*n*)**.

Exemplo do algoritmo de ordenação MERGE SORT em JavaScript:

```
let comps = 0, divisoes = 0, juncoes = 0;

function mergeSort(vetor) {

   if (vetor.length < 2) return vetor;

   let meio = Math.floor(vetor.length / 2);

   let vetEsq = vetor.slice(0, meio);


   let vetDir = vetor.slice(meio);

   divisoes++;


   vetEsq = mergeSort(vetEsq);
   vetDir = mergeSort(vetDir);

     let posEsq = 0, posDir = 0, vetRes = [];

   while (posEsq < vetEsq.length && posDir < vetDir.length) {
            comps++;
      if (vetEsq[posEsq] < vetDir[posDir]) {
         vetRes.push(vetEsq[posEsq]);
         posEsq++;
      }

      else {
         vetRes.push(vetDir[posDir]);
         posDir++;
      }
   }


   let sobra;

      if (posEsq < posDir) {
```

```
        sobra = vetEsq.slice(posEsq);
    }

    else {

        sobra = vetDir.slice(posDir);
    }

    juncoes++;


    return [...vetRes, ...sobra];
}

let nums = [77, 44, 22, 33, 99, 55, 88, 0, 66, 11];

let numsOrd = mergeSort(nums);

console.log({ numsOrd });
```

2.4 Quick Sort

O algoritmo de ordenação Quick Sort foi criado por C. A. R. Hoare em 1960, é considerado um dos métodos de ordenação mais rápidos, porém é um algoritmo por comparação não-estável.

Assim como o Merge Sort, o Quick Sort também usa a estratégia de “Divisão e Conquista”, a diferença é que o Quick Sort escolhe um elemento da lista como o “pivô(pivot)”, reorganiza o vetor/lista de modo em que todos os elementos a esquerda do pivô são menores do que ele, e todos os elementos a direita são maiores do que ele. No final da ação o pivô estará na sua posição final e existira duas sub lista a serem ordenadas, essa operação é conhecida como “partição”. E por último vem a recursividade, onde a sub lista da esquerda e a sub lista da direita são ordenadas.

No pior caso possui complexidade de tempo de execução: **O(n²)**, já no melhor caso possui a complexidade de tempo de execução: **O(n log n)**.

Exemplo do algoritmo de ordenação QUICK SORT em JavaScript:

```
let pass = 0, comps = 0, trocas = 0;
```

```
function quickSort(vetor, ini = 0, fim = vetor.length - 1) {


   if (fim <= ini) {
      return;     }

   pass++;

   const pivot = fim;
   let div = ini - 1;

   for (let i = ini; i < fim; i++) {
      comps++;
      if (vetor[pivot] > vetor[i] && i !== div) {
         div++;
         if (div !== 1) {
            [vetor[i], vetor[div]] = [vetor[div], vetor[i]];
            trocas++;
         }
      }
   }

   div++;

   comps++;
   if (vetor[div] > vetor[pivot] && div !== pivot) {
      [vetor[div], vetor[pivot]] = [vetor[pivot], vetor[div]];
      trocas++;
   }

   quickSort(vetor, ini, div - 1);

   quickSort(vetor, div + 1, fim);

}


let nums = [77, 44, 22, 33, 99, 55, 88, 0, 66, 11];

quickSort(nums);

console.log(nums);
```

3 Materiais e métodos ou desenvolvimento

O presente trabalho foi desenvolvido baseado em materiais bibliográficos e fontes da internet por meio do método do estudo analítico.

4 Resultados e discussão

Com isso chegamos a conclusão que para definir se um algoritmo é melhor do que o outro, deve se analisar a complexidade de tempo e espaço do algoritmo em realação a outro. No caso do Bubble Sort não é muito utilizado na prática como os outros algoritmos de ordenação, ele é mais usado para fins didáticos, é mais usado o Insertion Sort na prática do que o Bubble Sort quando se precisa ordenar um Array ou Lista com poucos dados, se for uma lista muito grande é melhor usar o Merge Sort ou o Quick Sort, ou qualquer outro algoritmo de ordenação tão rápido quanto o Quick Sort e Merge Sort ou até mais rápido, mas visando sempre usar aquele que usa pouca memória e ao mesmo tempo é rápido.

Considerações finais

Esse trabalho foi interessante por descobrir sobre a complexidade dos algoritmos, complexidade de tempo e espaço e a notação Big-O, para fazer análise de um algoritmo eficiente para situações diferentes, além de que existe diversos algoritmos de ordenação.

Referências

VIANA, Daniel. **Algoritmos de ordenação**. 2017. 7 f. TCC (Graduação) - Curso de Analise e Desenvolvimento de Sistemas, Fatec Franca - Faculdade de Tecnologia de Franca Dr Thomaz Novelino, São Paulo, 2021. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao. Acesso em: 18 nov. 2021.

GATTO, Elaine Cecília. **Algoritmos de Ordenação: Bubble Sort**. 2017. 10 f. Dissertação (Mestrado) - Curso de Analise e Desenvolvimento de Sistemas, Fatec Franca - Faculdade de Tecnologia de Franca Dr Thomaz Novelino, Franca, 2021. Disponível em: https://www.embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/. Acesso em: 21 nov. 2021.

PEREIRA, Wilder. **Introdução à Complexidade de Algoritmos**. 2019. 3 f. Dissertação (Mestrado) - Curso de Analise e Desenvolvimento de Sistemas, Fatec Franca - Faculdade de Tecnologia de Franca Dr Thomaz Novelino, Franca, 2021. Cap.

1. Disponível em: https://medium.com/nagoya-foundation/introdu%C3%A7%C3%A3o-%C3%A0-complexidade-de-algoritmos-4a9c237e4ecc. Acesso em: 24 nov. 2021.

BRUNET, João Arthur. **Ordenação por Comparação: Selection Sort**. 2019. 5 f. Dissertação (Mestrado) - Curso de Analise e Desenvolvimento de Sistemas, Fatec Franca - Faculdade de Tecnologia de Franca Dr Thomaz Novelino, Franca, 2021. Cap. 1. Disponível em: https://joaoarthurbm.github.io/eda/posts/selection-sort/. Acesso em: 24 nov. 2021.