Análise de Complexidade dos Algoritmos de Ordenação

1. Bubble Sort

Descrição: O Bubble Sort é um algoritmo de ordenação simples e popular, porém ineficiente – estudado apenas visando o desenvolvimento de raciocínio. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem.

Algoritmo:

```
public void bubblesort (){
  int LSup, i, j, temp;

  LSup = n-1;
  do{
    j = 0;
    for (i = 0; i < LSup; i++)
      if (vetor[i] > vetor[i+1]{
         temp = vetor[i];
         vetor[i] = vetor[i+1];
         vetor[i+1] = temp;
         j = i;
      }
      LSup = j;
  }while (LSup >= 1);
}
```

Notas:

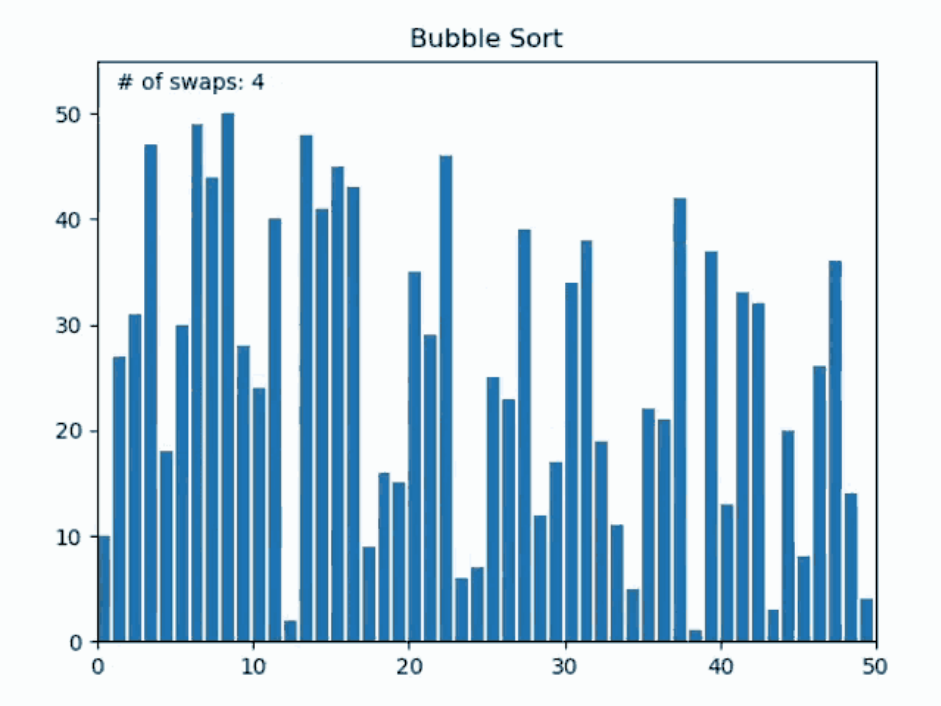
- ➔ LSup armazena o índice da última troca realizada durante a iteração atual do loop. Isso é útil porque, após cada iteração, todos os elementos à direita de aux estão garantidos como ordenados. Portanto, na próxima iteração, não há necessidade de verificar esses elementos novamente.
- ➔ Quando o algoritmo encontra dois elementos no vetor que estão fora de ordem, temp armazena temporariamente o valor de um dos elementos enquanto o outro é movido. Isso impede que um dos valores seja sobrescrito antes da troca ser completada.

Funcionamento:

- ➔ **Primeira Passagem:** Compara o primeiro elemento com o segundo. Se o primeiro é maior que o segundo, troca-os. Depois, compara o segundo com o terceiro e assim por diante. No final da passagem, o maior elemento "borbulha" para o final do vetor.

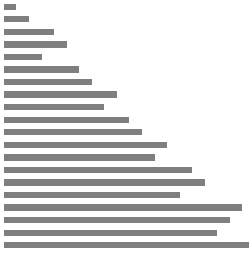
➔ **Segunda Passagem:** Repetimos o processo, mas agora não precisamos comparar o último elemento, pois ele já está no lugar certo. Então, comparamos os elementos até o penúltimo.

➔ **Repetições:** Continuamos assim até que nenhum elemento precise ser trocado, indicando que o vetor está ordenado.

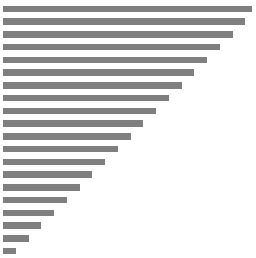


Análise de Complexidade:

- **Tempo de execução:**
  - **Melhor caso:** O melhor caso ocorre quando o vetor já está ordenado. No entanto, o algoritmo ainda percorre o vetor inteiro uma vez para verificar se nenhuma troca foi necessária. Portanto, o número de comparações é linear [C(n) = 0(n)] e o de movimentações é zero [M(n) = 0]. **O(n)**: Aparece quando você está lidando com uma única iteração do vetor
  - **O(n)**: Aparece quando você está lidando com uma única iteração do vetor.



- **Pior caso e Caso médio:** Tanto no pior quanto no médio caso, o número de comparações e de movimentações, [C(n) = (n²)] e [M(n) = (n²)], são quadráticos. Isso acontece porque, no pior caso, onde o vetor está completamente desordenado, o algoritmo precisa realizar o máximo de trocas e comparações possíveis, o que faz o número total de operações crescer quadraticamente com o tamanho do vetor. No caso médio, mesmo que o vetor não esteja completamente desordenado, o Bubble Sort ainda faz múltiplas passagens e comparações, resultando também em uma complexidade quadrática.
- **O(n²)**: Aparece quando você soma todas as comparações e trocas feitas ao longo de todas as iterações do do-while.



Se você somar todas as comparações feitas, você obtém uma série aritmética:

$$(n-1) + (n-2) + (n-3) + \ldots + 1$$

O total de comparações é dado por:

$$\frac{n(n-1)}{2}$$

Essa fórmula mostra que o número de comparações cresce quadraticamente com o tamanho do vetor, resultando na complexidade O(n²).

- **Espaço adicional:**
  - O Bubble Sort é um algoritmo de ordenação in-place, o que significa que ele não requer espaço adicional além do espaço necessário para armazenar o vetor. Portanto, a complexidade de espaço é [O (1)].

2. Quick Sort

Descrição: O Quick Sort é um algoritmo de ordenação que utiliza a técnica de divisão e conquista. Ele seleciona um "pivô" e particiona o vetor em dois sub-vetores: um com elementos menores que o pivô e outro com elementos maiores. Em seguida, o algoritmo é chamado recursivamente para ordenar os sub-vetores.

Algoritmo:

```
public void quicksort (){
   ordena (0, n-1);
}

private void ordena (int esq, int dir){
   int pivo, i = esq, j = dir, temp;

   pivo = vetor[(i+j)/2];
   do {
      while (vetor[i] < pivo)
         i++;
      while (vetor[j] > pivo)
         j--;
      if (i <= j) {
         temp = vetor[i];
         vetor[i] = vetor[j];
         vetor[j] = temp;
         i++;
         j--;
      }
   } while (i <= j);
   if (esq < j)
      ordena (esq, j);
   if (dir > i)
      ordena (i, dir);
}
```

Funcionamento:

- ➔ Escolher arbitrariamente um item do vetor como pivô;
- ➔ Percorrer o vetor a partir de seu início, até encontrar um item com chave maior ou igual à chave do pivô, cujo índice chamaremos de i;
- ➔ Percorrer o vetor a partir do final, até encontrar um item com chave menor ou igual à chave do pivô, cujo índice chamaremos de j;
- ➔ Trocar os itens v[i] e v[j];
- ➔ Continuar o percurso-e-troca até que os dois índices se cruzem.

Análise de Complexidade:

- **Tempo de Execução:**
  - **Melhor caso:** O melhor caso acontece quando, em todas as etapas, o vetor for dividido ao meio. Assim, o custo de comparações será $C(n) = 2C\left(\frac{n}{2}\right) + 2$, onde C(n/2) é o custo de ordenar cada metade e n é o custo de examinar cada item. Logo, $C(n) = 1.4 \log n$ e, em média, o tempo de execução será da ordem O (n logn).
  - **Pior Caso:** O pior caso acontece quando o pivô escolhido é um dos extremos de um conjunto de dados já ordenado. Neste caso, haverá n chamadas recursivas, eliminando um elemento por vez. Logo, necessita de uma pilha auxiliar para as chamadas recursivas de tamanho n e o número de comparações será $C(n) = n^2/2$.
  - **Caso Médio:** Na prática, o Quick Sort tende a ter um desempenho próximo ao caso médio de O (n log n), pois a escolha do pivô é frequentemente boa o suficiente para dividir o vetor de maneira equilibrada.

- **Espaço Adicional:**
  - O Quick Sort usa recursão, e o espaço de pilha necessário para a recursão pode chegar a O(log n) no melhor caso e até O(n) no pior caso. No entanto, como o algoritmo é in-place, ele não requer espaço adicional para armazenar o vetor além do que já é necessário, o que significa que a complexidade de espaço adicional para o vetor em si é O(1). Isso torna o Quick Sort eficiente em termos de uso de espaço.

3. Comparação dos Algoritmos

- **Eficiência de Tempo:** O Quick Sort é geralmente mais eficiente do que o Bubble Sort. A complexidade média de O(n log n) do Quick Sort é significativamente melhor do que a complexidade O(n²) do Bubble Sort, especialmente para vetores grandes. O Bubble Sort pode ser adequado para vetores muito pequenos ou para fins educacionais devido à sua simplicidade, mas não é prático para vetores grandes.

- **Eficiência de Espaço:** Ambos os algoritmos são eficientes em termos de espaço adicional. O Bubble Sort tem complexidade de espaço O(1), enquanto o Quick Sort tem O(log n) na média devido à recursão. Ambos são algoritmos in-place, o que significa que não requerem espaço adicional significativo além do necessário para armazenar o vetor.

4. Conclusão

O Quick Sort é geralmente mais rápido e eficiente em comparação com o Bubble Sort, especialmente para grandes conjuntos de dados, devido à sua complexidade de tempo média de O(n log n). O Bubble Sort, com sua complexidade de $O(n^2)$, não é prático para grandes vetores e é mais utilizado para fins educacionais e para aprender sobre algoritmos de ordenação básicos.