Filipe Bessa Carvalho, Jeann Victor Batista

Comparativo de Algoritmos de Ordenação por Ordem Alfabética

Breve comparativo entre algoritmos de ordenação de diferentes tipos: simples, ótimo e em tempo linear. Referente à disciplina DCE792.

Universidade Federal de Alfenas – UNIFAL-MG

Instituto de Ciências Exatas

Ciência da Computação – Bacharelado

Alfenas-MG, Brasil

27/11/2024

Sumário

Introdução

Um dos problemas mais extensivamente explorados da computação é a ordenação de um conjunto de dados em uma determinada ordem. Desde sua primeira concepção, a otimização e o aperfeiçoamento dos métodos de ordenação tem sido vista com extrema importância. O objetivo deste trabalho é comparar três algoritmos de ordenação em algumas áreas-chave. A proposta foi elaborada pelo professor Iago Augusto de Carvalho, como avaliação referente à disciplina de Algoritmos e Estruturas de Dados II, do Bacharelado em Ciência da Computação da UNIFAL-MG.

Os três algoritmos escolhidos encaixam-se cada um em uma categoria diferente. As categorias, orientadas pela complexidade do algoritmo, são: algoritmos de ordenação simples, ótimos e em tempo linear. Em linhas gerais, os algoritmos de ordenação simples possuem complexidade de tempo maior, e os metodos mais otimizados fazem uso da memória para manusear os dados. Essa prática reduz a complexidade de tempo, porém aumenta o gasto de memória.

Logo, uma comparação entre algoritmos desses três tipos deve levar em consideração alguns aspectos importantes (as áreas-chave mencionadas anteriormente). A comparação feita neste trabalho levou em consideração o tempo de execução, o número de operações de comparação, o número de operações de troca e o gasto de memória[1], como especificado na proposta do trabalho avaliativo.

Os três algoritmos escolhidos, seus funcionamentos, suas complexidades, o código utilizado e os resultados obtidos serão apresentados a seguir.

[1] Apenas espaços reservados para variáveis novas foram considerados

1 Algoritmos

Segue uma apresentação breve de todos os algoritmos usados neste trabalho, como operam e como foram implementados em linguagem c. Mais detalhes a respeito da implementação e os métodos de avaliação (contadores e temporizador) estão disponíveis em um repositório do github. Lá pode-se encontrar todo o código fonte, comentado extensivamente. Adicionalmente, releva-se que todas as comparações foram padronizadas usando a função *strcmp*, da biblioteca *string.h*.

1.1 Algoritmo simples

O algoritmo simples escolhido foi o *Bubble Sort*, criado em 1956, por Edward Harry Friend [1]. Nesse método, as comparações sempre serão feitas entre dados de índice adjacente. Inicia-se de uma extremidade, comparando cada valor com o próximo e, caso necessário para a ordenação, executando uma troca. Esse procedimento é repetido até chegar ao fim do vetor, quando as comparações são realizadas novamente a partir da extremidade inicial. O vetor é dado como ordenado quando, após realizar um percorrimento completo do vetor, nenhuma troca precisar ser feita.

Durante o ordenamento de um vetor, nota-se que os valores extremos (o maior ou o menor, dependendo da implementação) percorrem as casas por meio das trocas até chegarem na extremidade correta. Uma interpretação deste comportamento é que os números comportam-se como bolhas de ar na água, subindo à superfície uma por uma. A analogia das bolhas inspirou o nome *Bubble Sort*, "Ordenação Bolha"em inglês.

A seguir, uma imagem ilustrativa e o código em c utilizado durante a comparação (sem os contadores, para maior clareza).

[1] <https://en.wikipedia.org/wiki/Bubble_sort> (Acesso em 27/11/2024, às 16:47)

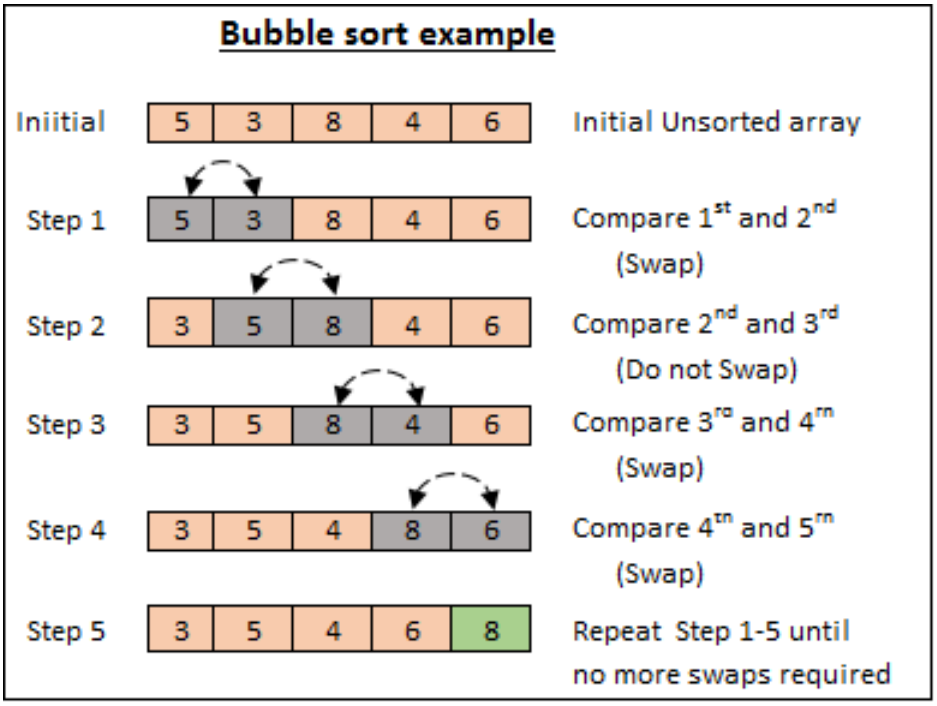

Figura 1 – Ilustração do Bubble Sort (fonte: Karuna Sehgal)

```
//ordena o vetor por meio do algorimo bubbleSort
void bubbleSort(tabela v[], int n){

    bool trocou;//registra se houve troca naquela iteracao
    for (int i = 0; i < n - 1; i++) {

        trocou = false;//falso no inicio da iteracao
        for (int j = 0; j < n - i - 1; j++) {
            if (strcmp(v[j].nome, v[j+1].nome) > 0) {
                swap(v, j, j+1);//troca os valores
                trocou = true;//houve troca
            }
        }
        if (trocou == false)
            break;//se nao houve nenhuma troca, saia
    }
}
```

1.2 Algoritmo ótimo

Os algoritmos ótimos são, em quase todos os casos, mais rápidos que os algoritmos simples. Ordenar um vetor com maior rapidez pode acelerar muitas partes do processamento. O algoritmo escolhido foi o *Quick Sort*, criado em 1960, por C. A. R. Hoare[2].

Este procedimento faz uso do princípio "dividir para conquistar". Ou seja, ele divide o vetor a ser ordenado em vetores cada vez menores, depois juntando-os. Existem três etapas básicas: a escolha do pivô, a divisão dos dados (em torno deste pivô) e a "quebra"do vetor (feita por uma chamada recursiva).

O pivô é a referência para a divisão da lista, já que o *Quick Sort* simplesmente coloca os valores menores que o pivô para um lado e os maiores para o outro (em cada chamada da função de repartição). Existem várias abordagens para a escolha do pivô, cada uma com suas vantagens e desvantagens. Escolher simplesmente o primeiro ou o último valor é simples, mas tem o pior caso quando o vetor já está ordenado. Escolher um elemento aleatório garante que não exista nenhum padrão para pior caso. Por fim, escolher um elemento mediano é mais complexo, mas tende a ser a abordagem ideal. No comparativo realizado, o último valor foi escolhido como pivô, mantendo a maior simplicidade possível.

Pode-se observar na ilustração abaixo que a repartição ocorre com a seleção do último valor como pivô. O vetor é dividido e os valores são organizados em torno do pivô até o último caso, onde só tem um valor remanescente. Quando esta situação é alcançada, observa-se que todos os valores estão ordenados em ordem crescente (ao ler os vetores de um único item da esquerda para a direita). Basta então retornar os valores nessa ordem para o vetor inicial. Nota-se que, ao criar chamadas recursivas e variáveis auxiliares adicionais, o gasto de memória aumenta, mas a rapidez consequente é bastante benéfica, principalmente ao tratar-se de conjuntos de dados extensos.

[2] <https://pt.wikipedia.org/wiki/Quicksort> (Acesso em 27/11/2024, às 17:53)

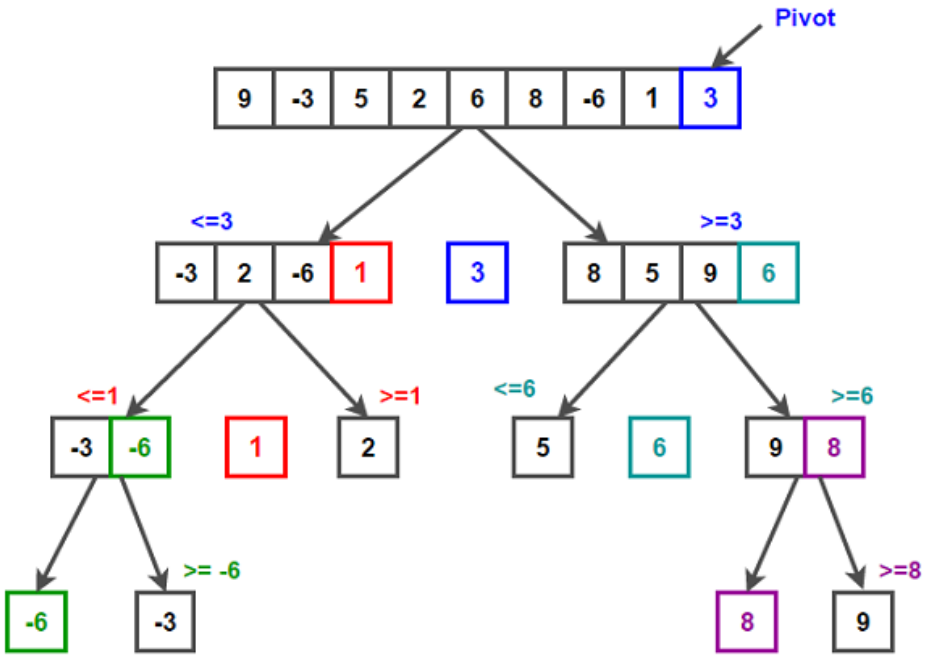

Figura 2 – Ilustração do Quick Sort (fonte: <https://www.techiedelight.com/es/quicksort/> Acesso em 27/11/2024, às 17:10)

Seguem as duas funções da implementação em c utilizada.

```
//reparte o vetor e separa eles em torno do pivo
int reparticao(tabela v[], int inf, int sup) {

    char* pivo = v[sup].nome;//ultimo valor e pivo
    int i = inf - 1;
    for (int j = inf; j <= sup - 1; j++) {
        if (strcmp(v[j].nome, pivo) < 0) {
            i++;
            swap(v, i, j);
        }
    }
    swap(v, i+1, sup);
    return i + 1;
}

//ordena o vetor por meio do algoritmo quickSort
void quickSort(tabela v[], int inf, int sup) {

    if (inf < sup) {
```

```
        int ir = reparticao(v, inf, sup);

        quickSort(v, inf, ir - 1);
        quickSort(v, ir + 1, sup);
    }
}
```

1.3 Algoritmo em tempo linear

O grande diferencial dos algoritmos de ordenação em tempo linear é que, como o nome indica, a complexidade de tempo da ordenação é linear: proporcional ao número de elementos na lista. O algoritmo desta classe escolhido foi o *Bucket Sort*.

O *Bucket Sort* faz uso de "buckets", baldes em inglês, para auxiliar no ordenamento. Baldes são recipientes de dados (espaços reservados na memória) que são preenchidos de acordo com algum critério. No caso do ordenamento em ordem alfabética, cada balde pode ser preenchido com base na primeira letra do nome (ignorando acentuação, existem então 26 baldes, de "A"até "Z").

Depois de separar os dados em baldes diferentes, cada um desses baldes é ordenado individualmente por meio de algum outro algoritmo (na implementação usada, este algoritmo foi o *Quick Sort*). Depois disso, o vetor final pode ser montado, colocando os itens de um balde seguido do outro, na ordem correta.

Por exemplo, os baldes separados pela primeira letra do nome são preenchidos e ordenados. O vetor final é o balde de todos os nome que começam com a letra "A"em ordem seguido do balde correspondente à letra "B", "C"e assim por diante.

A criação de baldes tem um custo de memória bastante elevado. Não obstante, em conjuntos muito grandes de dados e em sistemas onde não há uma limitação rígida de memória, o *Bucket Sort* e outros algoritmos de ordenação em tempo linear podem ser bastante desejáveis e se destacar pela rapidez do processamento.

Seguem uma imagem ilustrativa e a implementação em c. O algoritmo de *Quick Sort* usado é o mesmo da seção 1.2.

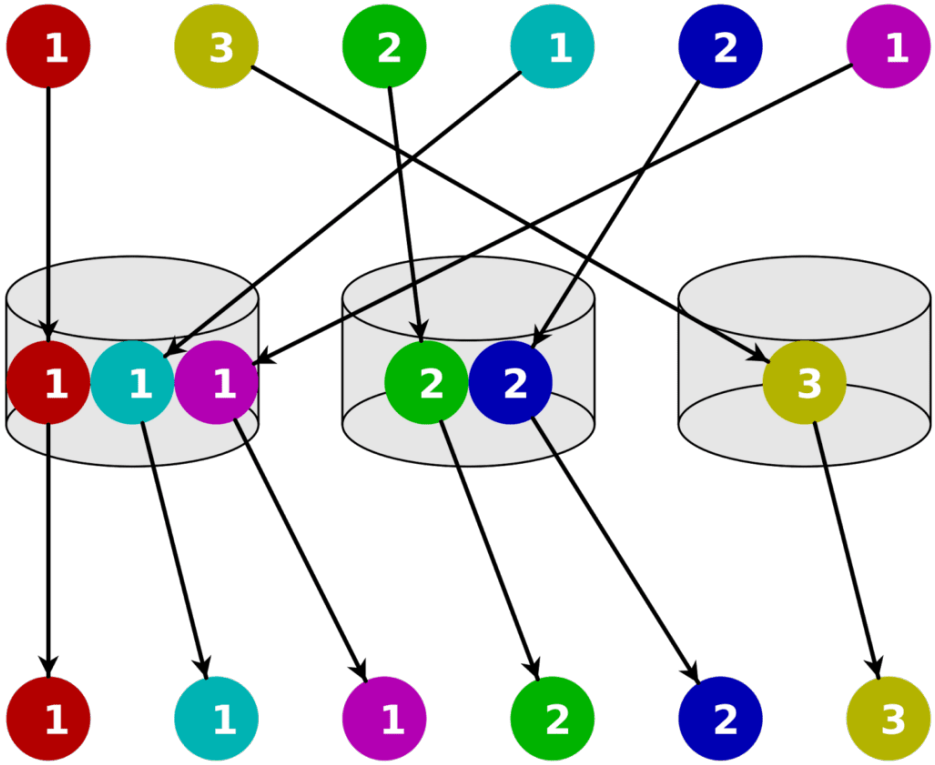

Figura 3 – Ilustração do Bucket Sort (fonte: Karuna Sehgal)

```
void bucketSort(tabela *v, int n) {

    int nBucket = 26;
    tabela **buckets = malloc(nBucket * sizeof(tabela*));
    int *tamBucket = calloc(nBucket, sizeof(int));

    tabela *espBucket = malloc(n * nBucket * sizeof(tabela));
    for (int i = 0; i < nBucket; i++) {
        buckets[i] = espBucket + i * n;
    }

    for (int i = 0; i < n; i++) {
        int iBucket = 0;
        char c = v[i].nome[0];

        /*O condicional abaixo define o indice do balde com base
        na primeira letra do nome
        */
        if(c >= 'A' && c <= 'Z')
            iBucket = c - 'A';
```

```
        else if(c >= 'a' && c <= 'z')
            iBucket = c - 'a';

        /* -buckets so todos os baldes;
            -iBucket e o balde da insercao
            -tamBucket e o indice da ultima posicao do balde
            -v[i] e o nome
        */
        buckets[iBucket][tamBucket[iBucket]++] = v[i];
    }

    int index = 0;
    for (int i = 0; i < nBucket; i++) {
        if (tamBucket[i] > 1) {
            quickSort(buckets[i], 0, tamBucket[i] - 1);          }
        memcpy(v+index,buckets[i],tamBucket[i]*sizeof(tabela));
        index += tamBucket[i];
    }

    free(espBucket);
    free(buckets);
    free(tamBucket);
}
```

2 Resultados

2.1 Dados obtidos

Os algoritmos tiveram alguns aspectos importantes rastreados e comparados, com o intuito de descobrir quais os pontos fortes e fracos de cada um. Essas informações podem ser usadas em conjunto com o conhecimento das limitações e necessidades do projeto desenvolvido para escolher o algoritmo mais adequado para determinada situação.

Como mencionado anteriormente, as quatro medidas que fazem parte deste comparativo são: tempo de execução, gasto de memória, número de trocas e número de comparações. Os algoritmos foram todos executados na mesma máquina.

Gráficos foram criados para a melhor compreensão. Note que o eixo y faz uso de escala logaritmica. Esta decisão foi tomada para facilitar a visualização das diferenças.

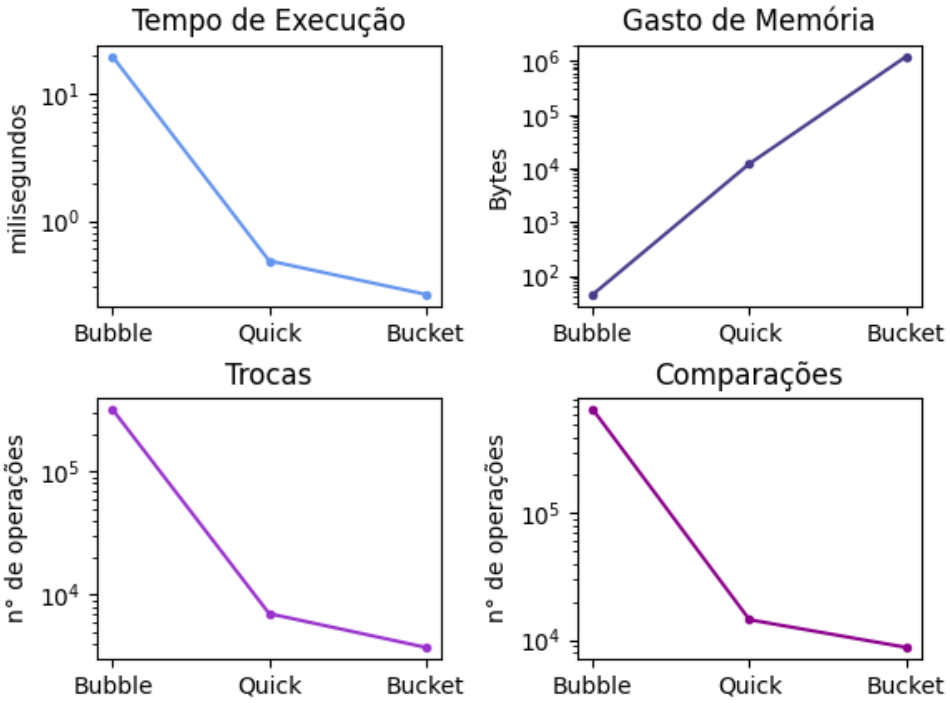

Figura 4 – Gráfico criado por meio da biblioteca *Matplotlib*

Estes dados serão discutidos no capítulo seguinte, juntamente com um levantamento das vantagens e desvantagens de cada algoritmo.

2.2 Considerações finais

Ao analisar os dados, pode-se notar claramente as tendências do problema de ordenação. Um tema importante é o aumento do gasto de memória com intuito de melhorar o manuseio dos dados e assim reduzir as operações e o tempo de execução. A seguir, uma lista com os algoritmos avaliados e suas principais informações.

- *Bubble Sort*
  - Possui a maior complexidade de tempo, $O(n^2)$.
  - Tem o menor gasto de memória. O gasto está puramente relacionado a variáveis auxiliares de índice, lógicas e temporárias usadas durante as trocas.
  - Tem o maior número de comparações e trocas. A simplicidade do algoritmo significa que o manuseio dos dados não é tão eficiente quanto possível.
  - A implementação em código foi a mais simples.

Em suma, o algoritmo não é eficiente para conjuntos de dados grandes, já que a complexidade cresce de forma exponencial. Não obstante, o *Bubble Sort* e outros algoritmos simples podem ser considerados para atividades didáticas, listas muito pequenas e aplicações com restrições extremas de memória. Um exemplo destas aplicações é dentro do *Bucket Sort*. Quando os baldes têm tamanho pequeno, em algumas implementações, algoritmos como o *Insertion Sort* (um algoritmo simples) são usados.

- *Quick Sort*
  - Possui complexidade de tempo logaritmica $O(n \cdot \log n)$, resultando em um tempo de execução muito menor que o algoritmo simples.
  - O crescimento do gasto de memória é consequência da criação de novas variáveis, como, por exemplo, a variável usada para armazenar o nome que será usado de pivô.
  - O número de comparações e trocas caiu drasticamente ao comparar-se com o *Bubble Sort*, devido ao manuseio eficiente dos dados, seguindo a filosofia de "dividir para conquistar."
  - A implementação em código foi ligeiramente mais complexa por fazer uso de chamadas recursivas.

Em suma, o algoritmo ótimo se destacou no caso específico usado nesta comparação, já que consegue ser claramente mais rápido que o algoritmo simples, e além disso, não se mantém atrás da ordenação em tempo linear por uma margem grande, mesmo consumindo muito menos memória. Pode-se dizer que este algoritmo consegue obter um equilíbrio muito bom entre as medidas tomadas.

- *Bucket Sort*
  - Possui complexidade de tempo linear $O(n + k)$. O tempo de execução foi o menor dentre todos os métodos.
  - O gasto de memória é consideravelmente maior para conseguir abarcar o espaço necessário para todos os baldes e suas respectivas informações.
  - O número de comparações e trocas foi o menor do comparativo.
  - A implementação em código foi a mais complexa.

Em suma, o algoritmo em tempo linear se destacou por ser o mais rápido e realizar o menor número de trocas. Porém, a melhoria de performance foi pequena ao ser comparada com o salto entre o algoritmo simples e o algoritmo ótimo. Uma consideração importante é que, devido ao tamanho do conjunto de dados, este algoritmo não demonstrou ser o mais desejável, podendo ser mais adequado para conjuntos de dados mais extensos.

Com os pontos trazidos acima, espera-se que os principais pontos fortes e fracos, as vantagens e desvantagens de cada algoritmo tenham ficado claras, e, adicionalmente, as condições que definem qual tipo de algoritmo é o mais adequado para a implementação.

Comparativos como este auxiliam na escolha do melhor método, resultando em *software* mais eficiente. Nem sempre o algoritmo mais rápido, com menor complexidade de tempo e número de operações será o melhor para determinada aplicação.