Algoritmos de ordenação – *Bubble Sort, Insertion Sort e Selection Sort.*

André Neves Medeiros, 2020.1.08.028 – Universidade Federal de Alfenas, 2020.

Relatório apresentado à disciplina de Algoritmos e Estrutura de Dados I, orientada pelo professor Paulo Alexandre Bressan na Universidade Federal de Alfenas em Minas Gerais.

Introdução e objetivo

O objetivo geral de um algoritmo de ordenação é ler, ordenar (de forma crescente) e apresentar os mesmos valores organizados para o usuário.

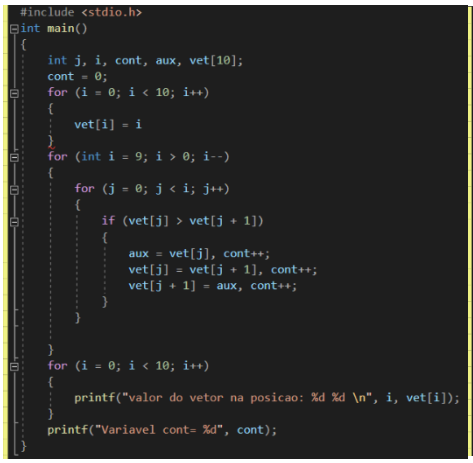
O *Bubble Sort* é um algoritmo de ordenação que percorre o vetor várias vezes, a cada "passada" pelo vetor mais organizado ele fica, isso porque a ideia do algoritmo é flutuar o maior elemento da sequência. Tem esse nome por conta de o comportamento ser parecido com o de como bolhas na água, as bolhas procuram seu próprio nível. Entretanto esse algoritmo não é ideal para programas que exijam velocidade, pelo motivo de que pode ser necessário várias passagens pelas mesmas posições dos vetores, isso significa que precisa de mais processamento (por passar várias vezes pelo vetor).

O *Insertion Sort* é um algoritmo que é eficiente sobretudo em vetores pequenos (conjunto de dados pequeno), diferente do anterior, esse algoritmo só requer uma passada pelo vetor, e enquanto passa vai ordenando os valores à sua esquerda. Sendo o anterior maior que o atual, ele troca e compara com o anterior a ele, até que a comparação (vet[i] < vet[i-1]) seja falsa. O vetor não altera a ordem de valores iguais. Outra vantagem desse algoritmo é que ele pode funcionar perfeitamente em tempo real, sendo os valores organizados conforme a entrada de dados.

O *Selection Sort* é outro algoritmo de ordenação, a diferença dele para os demais é que é passado sempre o menor valor para o início do vetor, em seguida o segundo menor valor para a segunda posição do vetor e assim por diante até que os valores estejam ordenados no vetor.

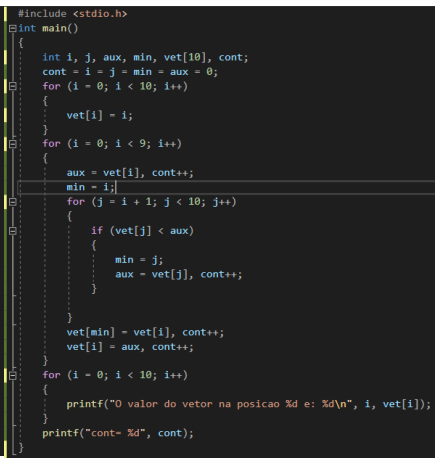
As entradas foram padronizadas de três diferentes formas: ordenado (os valores de entrada já vinham ordenados de forma crescente), desordenado (valores de entrada embaralhados aleatoriamente (foi feita uma média de 3 execuções do código) e inversamente ordenado (entrada em ordem decrescente)).

Os algoritmos estão respectivamente dispostos abaixo.



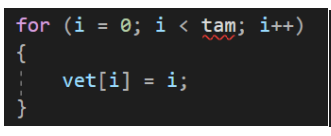
```c
#include <stdio.h>
int main()
{
    int j, i, cont, aux, vet[10];
    cont = 0;
    for (i = 0; i < 10; i++)
    {
        vet[i] = i
    }
    for (int i = 9; i > 0; i--)
    {
        for (j = 0; j < i; j++)
        {
            if (vet[j] > vet[j + 1])
            {
                aux = vet[j], cont++;
                vet[j] = vet[j + 1], cont++;
                vet[j + 1] = aux, cont++;
            }
        }
    }
    for (i = 0; i < 10; i++)
    {
        printf("valor do vetor na posicao: %d %d \n", i, vet[i]);
    }
    printf("Variavel cont= %d", cont);
}
```

```c
#include<stdio.h>
int main()
{
    int vet[10];
    int i, j, aux, cont;
    cont = 0;
    for (i = 0; i < 10; i++)
    {
        printf("Digite o valor do vetor na posicaAao %d: \n", i);
        scanf_s("%d", &vet[i]);
    }
    for (int i = 1; i < 10; i++)
    {
        aux = vet[i], cont++;
        for (j = i - 1; j >= 0 && vet[j] > aux; j--)
        {
            vet[j + 1] = vet[j], cont++;
        }
        vet[j + 1] = aux, cont++;
    }

    for (int i = 0; i < 10; i++)
    {
        printf("valor do vetor na posicao %d: %d\n", i, vet[i]);
    }
    printf("Movimentações = %d", cont);
}
```

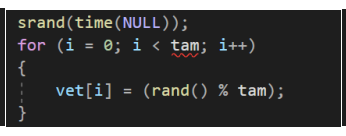


```c
#include <stdio.h>
int main()
{
    int i, j, aux, min, vet[10], cont;
    cont = i = j = min = aux = 0;
    for (i = 0; i < 10; i++)
    {
        vet[i] = i;
    }
    for (i = 0; i < 9; i++)
    {
        aux = vet[i], cont++;
        min = i;
        for (j = i + 1; j < 10; j++)
        {
            if (vet[j] < aux)
            {
                min = j;
                aux = vet[j], cont++;
            }

        }
        vet[min] = vet[i], cont++;
        vet[i] = aux, cont++;
    }
    for (i = 0; i < 10; i++)
    {
        printf("O valor do vetor na posicao %d e: %d\n", i, vet[i]);
    }
    printf("cont= %d", cont);
}
```

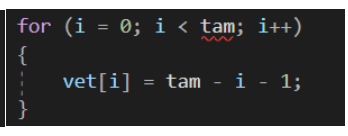
Para a sequência desordenada foi necessária a inclusão de outras duas bibliotecas, a fim de otimizar os testes com todos os algoritmos, essas foram: “stdlib.h” e “time.h”, além é claro da “stdio.h”; os valores de entrada são pertencentes aos intervalos [0, 10[; [0, 100[; e [0,1000[. Ademais foi necessária alteração das primeiras linhas de comando, sendo que a leitura de sequência ordenada, desordenada e inversa estão representadas respectivamente por (“tam” é o tamanho do vetor):



```c
for (i = 0; i < tam; i++)
{
    vet[i] = i;
}
```



```c
srand(time(NULL));
for (i = 0; i < tam; i++)
{
    vet[i] = (rand() % tam);
}
```



```c
for (i = 0; i < tam; i++)
{
    vet[i] = tam - i - 1;
}
```

Para os valores desordenados, onde se lê 100 é um representante do tamanho do vetor, sendo um vetor [10], o valor máximo é 9.

Resultados

Os resultados tabulados são produtos da inserção de uma variável inteira contadora (“cont”), inserida em pontos específicos a fim de contar todas as movimentações de cada setor do vetor em todos os 3 diferentes tipos de sequência (ordenado, desordenado e inversamente ordenado). O intuito disso é verificar qual algoritmo é mais eficiente (resultado/número de trocas), afinal quanto menos os valores se movimentarem menos processamento é necessário e mais rápido é o programa.

As sequencias ordenadas usadas foram o conjunto dos naturais até o valor de posição igual ao índice máximo do vetor, por exemplo, para o vetor [10], foram usados os números {0, 1, 2, 3, ..., 9.}. O inverso foi aplicado ao “inversamente ordenado”.

[10]

Algoritmos Entradas/	*Bubble Sort*	*Insertion Sort*	*Selection sort*
Ordenado	0	18	27
Desordenado	56	40	40
Inversamente ordenado	135	63	52

[100]

Algoritmos Entradas/	*Bubble Sort*	*Insertion Sort*	*Selection sort*
Ordenado	0	198	297
Desordenado	6799	2495	601
Inversamente ordenado	14850	5148	2797

[1000]

Algoritmos Entradas/	*Bubble Sort*	*Insertion Sort*	*Selection sort*
Ordenado	0	1998	2997
Desordenado	738899	249694	8232
Inversamente ordenado	1498500	501498	252997

Conclusão

Conclui-se, portanto, que o algoritmo *Bubble Sort* para sequencias já ordenadas, independentemente do tamanho do vetor é a melhor opção por exigir menos processamento, na verdade, a movimentação de valores do vetor nesse caso é nula. Entretanto para vetores inversamente ordenados, teve o pior resultado em todos os tamanhos vetoriais, com mais de um milhão de ações necessárias para ordená-lo no último caso.

Selection e *Insertion* são ambos melhores que o *Bubble* em vetores desordenados e inversamente ordenados em todos os tamanhos; entre eles, *Selection* é a melhor opção para vetores desordenados e inversamente ordenados maiores ou iguais a 10. O *Insertion* só vai ser melhor que *selection* em organizar sequencias já ordenadas de qualquer tamanho, entretanto como já dito anteriormente, sequências ordenadas têm custo 0 de processamento (troca de valores dentro do vetor) para o primeiro algoritmo, o *Bubble Sort*.