

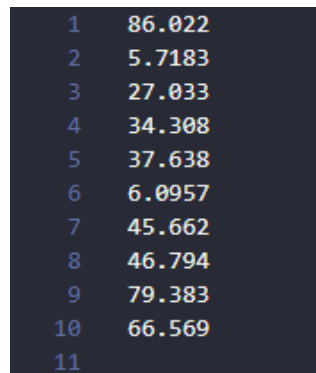
1 Introdução

Esse trabalho é a resolução proposta para o Exercício de Programação 1, da matéria de Organização e Arquitetura de Computadores II (ACH2055), ministrada pelo Prof. Dr. Clodoaldo A. de Moraes Lima.

Todos os códigos desenvolvidos podem ser vistos no repositório do GitHub, na url <https://github.com/victorIsDeving/sistemas-de-informacao/tree/main/oac2-ACH2055/ep1>.

2 Objetivo

O trabalho visa utilizar a linguagem MIPS Assembly para criar um programa que receba como *input* um arquivo de texto *.txt* com uma lista de números e gere um *output* dos números desta lista ordenados. Um exemplo de *input* é mostrado na Figura 1.



```
1 86.022
2 5.7183
3 27.033
4 34.308
5 37.638
6 6.0957
7 45.662
8 46.794
9 79.383
10 66.569
11
```

Figura 1: Exemplo de um arquivo de texto que pode ser usado como *input*.

O *output* também deve ser escrito no mesmo arquivo de texto utilizado como *input*, depois da lista original.

Dois métodos de ordenação devem ser implementados, um com complexidade $O(n^2)$ e outro com $O(n \log n)$.

3 Metodologia

O exercício foi com o programa MARS (*MIPS Assembler and Runtime Simulator*[6]), disponibilizado pela Universidade do Estado do Missouri. É uma IDE feita para programar em MIPS Assembly, com propósitos educacionais.

Os métodos de ordenamento feitos para esse trabalho, de acordo com o proposto pelo professor, são o *Bubble Sort* e o *Quick Sort*.

4 Desenvolvimento

O exercício foi feito em partes, descritas nos itens abaixo, que depois foram juntadas em um único arquivo, o EP1-SORTING.ASM, que é a resolução do exercício.

4.1 Leitura de um arquivo de texto

O primeiro passo tomado foi o desenvolvimento de um programa que faça a leitura do texto de um arquivo e imprima esse texto no console.[2] O programa é o READ-PRINT.ASM. Ele recebe o caminho para um arquivo de texto, lê o arquivo, armazena a string lida em um registrador e imprime o conteúdo desse registrador no console do MARS.

4.2 Escrita em um arquivo de texto

O segundo passo foi desenvolver um programa que escreva em um arquivo de texto. [3] O programa é o READ-WRITE.ASM. A ideia dele é a de inserir uma string pré definida no arquivo original. Começa com o mesmo processo de leitura de um arquivo realizado no programa READ-PRINT.ASM. Depois, copia-se, byte a byte, o conteúdo dessa string para a string final, adicionando quebras de linha para a divisão do conteúdo original para o conteúdo adicionado. Então, começa a copiar a string pré definida para a string final, de novo byte a byte, começando a inserir os caracteres imediatamente depois das quebras de linha. Terminando com o *syscall* de escrita de arquivo com a string final que concatena o conteúdo lido, quebras de linha e o conteúdo pré definido para inserção, passando de forma "hard coded" a quantidade total de caracteres que seria inserida no novo arquivo.[5]

4.3 Quantidade de números em uma string

O terceiro passo foi desenvolver um programa para verificar o tamanho de uma string de números. O programa é o ARRAY-SIZE.ASM. Esse programa passa por cada caractere do arquivo lido e verifica, byte a byte, qual é um espaço, que é o caractere esperado para separar um número de outro no arquivo de texto que será lido. Para cada espaço encontrado ele conta um número no arquivo, retornando o total de número no final da leitura.

Para essa parte teve o auxílio de uma tabela ASCII de caracteres para encontrar qual o número decimal representativo dos espaços.[1]

4.4 Transformar conteúdo lido em uma array de floats

O quarto passo foi desenvolver um programa para converter a string com uma lista de números em uma array de floats. O programa é o GET-FLOATS.ASM.

O método utilizado para essa transformação é byte a byte de cada caractere do arquivo, com o auxílio de uma tabela ASCII de caracteres para decimais. [1] O método inicia com dois ponteiros no caractere anterior ao primeiro dígito do número, no caso do primeiro número da lista seria um NUL, ASCII 0, em qualquer outro número da lista é um espaço, ASCII 32. Basicamente, o número é dividido em duas partes, a que identifica a parte inteira do número e a que identifica a parte decimal do número.

Inicia-se com uma busca pelo ponto, ASCII 46, que separa a parte inteira da parte decimal do número em questão. Encontrado o ponto, um dos ponteiros vai ficar no ponto e o outro continua no caractere anterior ao primeiro dígito do número. Daí, começa um loop que vai iterar do número menos significativo da parte inteira (casa das unidades) até o número mais significativo da parte inteira (casa das dezenas, centenas, milhares, essa pode variar), ou seja, vai iterar até que o ponteiro que está no ponto dos decimais chegue no ponteiro antes do primeiro caractere. Então o loop pega o byte ASCII do número, passa por uma condicional parecido com um *switch* que compara com cada ASCII de 0 a 9 para identificar qual o dígito em questão. Para identificar o hexadecimal correto para a float de cada dígito foi usado um conversor online. [4] Em cada iteração também tem um registrador que indica a potência de 10 para multiplicar o dígito encontrado, na primeira iteração o dígito é multiplicado por 10^0 , o segundo por 10^1 , o terceiro por 10^2 e assim por diante. Cada resultado encontrado depois da multiplicação é somado em um registrador separado, que vai guardar o número em si que vai para a array de floats.

A leitura da parte decimal segue o mesmo raciocínio, mas agora enquanto um ponteiro aponta para o ponto que separa a parte inteira da parte decimal o outro ponteiro vai apontar para o espaço, ASCII 32, no final do número, o primeiro caractere depois do final do número. A iteração será do número mais significativo para o menos significativo da parte decimal, até ambos os ponteiros apontarem para o espaço depois do número. Da mesma forma que na parte inteira, cada dígito passa por uma condicional tipo *switch* para identificar qual o dígito, a diferença é que, depois de identificar o número, multiplica-se por uma potência negativa de 10. O primeiro número é multiplicado por 10^{-1} , o segundo é multiplicado por 10^{-2} , o terceiro por 10^{-3} e assim por

diante. O resultado dessa multiplicação é somado no registrador que contém o número resultado que vai para a array.

Montado o número através dessas somas, ele é colocado em uma array resultado que terá os números da lista convertidos para *floats*.

Uma observação: na construção da parte decimal do número, pode ocorrer uma imprecisão na multiplicação pelas potências de 10 negativas. Por exemplo, ao realizar a multiplicação $2 * 10^{-2}$ o resultado será 0.0199. Não foi possível resolver esse problema ocasional de imprecisão.

4.5 Bubble Sort

O quinto passo foi desenvolver um programa para ordenar uma array de *floats* pela técnica do *Bubble Sort*. O programa é o BUBBLE-SORT.ASM.

Uma ordenação simples por *Bubble Sort*, que foi inspirado em um código em C que foi desenvolvido no curso de Introdução à Análise de Algoritmos, com o Prof. Dr. Márcio Moreto.

4.6 Quick Sort

O sexto passo foi desenvolver um programa para ordenar uma array de *floats* pela técnica do *Quick Sort*. O programa é o QUICKSORT.ASM.

O código em MIPS para o *Quick Sort* foi inspirado em um código online. [7]

5 Conclusões

O código é funcional e ordena corretamente os números apresentados na lista. Contudo dois problemas foram encontrados e não foi possível a sua resolução, a imprecisão na multiplicação por potências negativas de dez e a escrita de floats em um arquivo de texto.

Quanto aos métodos de ordenação, como esperado, o quicksort se mostra mais eficiente na ordenação de listas maiores, mas não tem muita diferença na velocidade quando as listas são pequenas. O gráfico apresentado na Figura 2 mostra a comparação entre as velocidades de ordenação para listas de tamanho 10, 30, 50 e 100. A Tabela 1 mostra os tempos de execução obtidos com um teste simples de *syscall* com código 30, usando o resultado de *low order* como base para a comparação. [5]

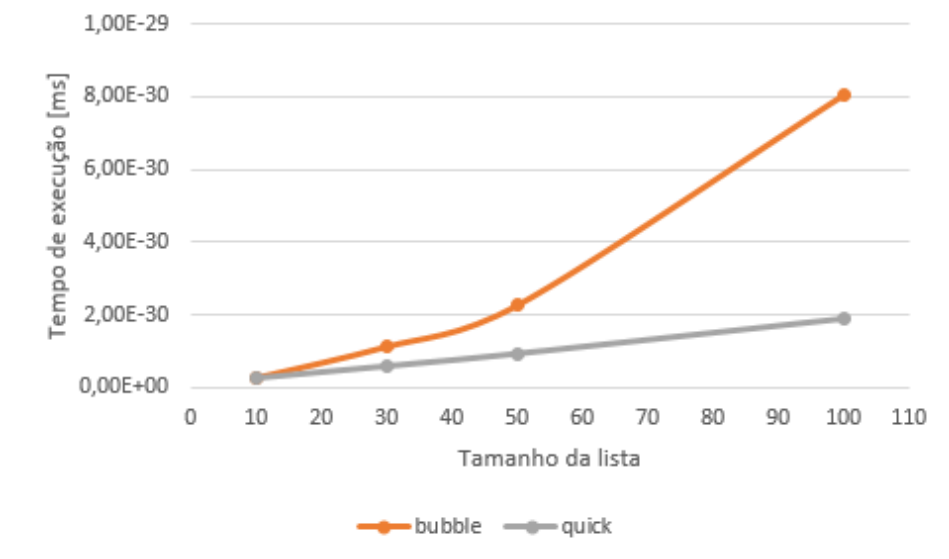


Figura 2: Gráfico comparando velocidades para ordenação de listas de diferentes tamanhos.

Tamanho	Bubble Sort	Quick Sort
10	2,74E-31	2,61E-31
30	1,13E-30	5,89E-31
50	2,27E-30	9,34E-31
100	8,04E-30	1,90E-30

Tabela 1: Tabela com os tempos de execução obtidos, em milissegundos, nos testes de velocidade.

Referências

- [1] Inc. Appropriate Solutions. *Decimal ASCII Chart*. URL: <https://asciichart.com/>.
- [2] Oliba! DesCOMplica. *Programação em Assembly MIPS - Aula 18 - Manipulação de Arquivos Texto*. URL: https://www.youtube.com/watch?v=WFb0xwY8bNU&ab_channel=DesCOMplica%2C0liba%21.
- [3] Oliba! DesCOMplica. *Programação em Assembly MIPS - Aula 19 - Manipulação de Arquivos Texto (Escrita de Dados)*. URL: https://www.youtube.com/watch?v=pCEjwgpx8ik&ab_channel=DesCOMplica%2C0liba%21.
- [4] h-schmidt. *IEEE-754 Floating Point Converter*. URL: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.
- [5] Otterbein student Tony Brock. *SYSCALL functions available in MARS*. URL: <http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html>.
- [6] Missouri State University. *MARS (MIPS Assembler and Runtime Simulator)*. URL: <http://courses.missouristate.edu/kenvollmar/mars/>.
- [7] Unknown. *Quick sort implementation in MIPS assembly*. URL: <https://www.programminghomeworkhelper.com/quick-sort-implementation-in-mips-assembly/>.