CIÊNCIA DA COMPUTAÇÃO

Algoritmos de ordenação na linguagem de programação Assembly

Victor Emmanuel Susko Guimarães
Cristiano Augusto Dias Mafuz

I. Introdução

A arquitetura de um computador diz respeito à forma como os diversos componentes do computador são organizados, e determina aspectos relacionados ao conjunto de instruções que o computador opera, permitindo a comunicação entre hardware e software.

Nesse contexto, seguem algumas explicações à respeito da arquitetura de computadores:

ISA (*Instruction Set Architecture*) de um processador é o documento que define o conjunto de instruções que ele pode executar, o tamanho das instruções, o número de registradores, o tipo das instruções, como essas instruções são organizadas na memória e como o processador lê e executa as instruções.

RISC (*Reduced Instruction Set Computer*) é uma linha de arquitetura que favorece um conjunto simples e pequeno de instruções que levam aproximadamente a mesma quantidade de tempo para serem executadas.

MIPS (*Microprocessor without interlocked pipeline stages*) é uma arquitetura de microprocessador do tipo RISC, que possui uma ISA própria, a qual será explorada neste trabalho.

ASSEMBLY é uma linguagem de programação em que cada instrução corresponde a uma operação simples do processador, como somar, mover ou comparar valores. A linguagem assembly é específica para cada arquitetura de computador, e é usada para programar dispositivos como microprocessadores e microcontroladores.

Dadas as importantes definições, o objetivo deste trabalho é construir dois algoritmos de ordenação diferentes, utilizando a linguagem de programação assembly em um simulador de MIPS, sendo que o primeiro algoritmo é o *Selection Sort* e o segundo algoritmo é o *Insertion Sort*.

Das disposições do trabalho, a seção II possui o referencial teórico relativo às funcionalidades utilizadas no simulador para a solução dos algoritmos e as explicações quanto aos métodos de ordenação, a seção III aborda os procedimentos da construção dos algoritmos e as métricas avaliativas dos testes, a seção IV traz os resultados dos testes segundo a métrica estipulada, bem como gráficos que avaliam os resultados dos algoritmos e, por fim, a seção V mostra uma síntese geral de todo o conteúdo apresentado, juntamente com uma análise crítica dos resultados.

II. Referencial Teórico

Primeiramente, é fundamental compreender o que é um algoritmo de ordenação e como os algoritmos propostos são executados, a fim de estabelecer uma comparação das características entre ambos. Em seguida, é necessário mostrar quais instruções o simulador consegue operar e como elas funcionam para que os algoritmos consigam ordenar corretamente os números.

Na computação, as ordenações são fundamentais em muitos sistemas digitais, pois permitem que um conjunto de elementos (números ou palavras) seja ordenado conforme algum parâmetro. Por isso, vários métodos de ordenação foram desenvolvidos ao longo do tempo, visando otimizar o tempo gasto e o espaço necessário para que esses números possam ser organizados.

Um método de ordenação **estável** é aquele que mantém a ordem relativa de elementos com o mesmo valor. Por exemplo, a figura 1.1 mostra um vetor de 5 posições em que dois elementos têm o mesmo valor, que no caso é o número 2. Para distinguir ambos, a figura 1.2 mostra que o elemento 2 roxo vem antes do elemento 2 vermelho. Se um método de ordenação estável fosse aplicado para ordenar este vetor, tem-se que o primeiro 2 (roxo) virá antes do segundo (vermelho), como mostrado na figura 3.

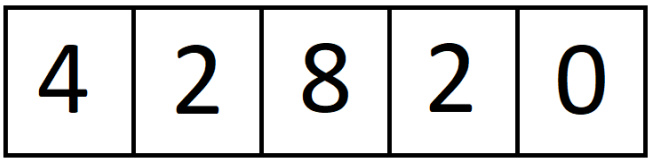


Figura 1.1 - vetor desordenado

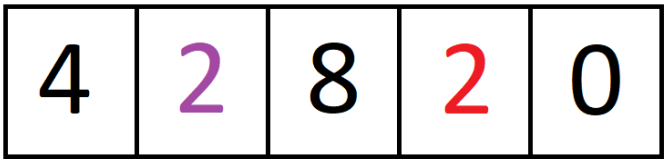


Figura 1.2 - vetor desordenado com elementos repetidos coloridos

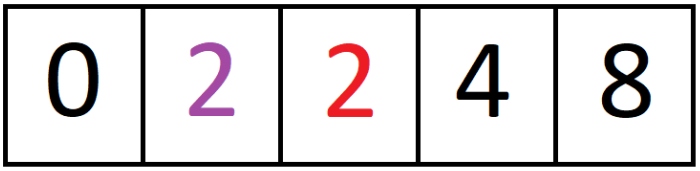


Figura 1.3 - vetor ordenado de maneira estável

Além disso, dizer que um método de ordenação é ***in situ*** significa que não foram criados outros vetores, chamados de subvetores, que auxiliam na ordenação. Logo, um método que não é *in situ* possui mais custo em relação ao espaço de memória do que um método que possui ordenação *in situ*. A figura 2 mostra um exemplo de método de ordenação que não é *in situ*.

Figura 2 - ordenação não *in situ*

A **ordem de complexidade** de um algoritmo é uma forma de mensurar como o tempo e espaço de memória gastos variam em função do tamanho da entrada. A ordem de complexidade é representada por uma notação matemática chamada O grande ou Big O, que denota qual o limite superior do crescimento da função que descreve o algoritmo. Por exemplo, se um algoritmo tem uma ordem de complexidade de O(n), isso significa que o tempo de execução é proporcional ao tamanho da entrada n. Da mesma forma, um algoritmo que tem uma ordem de complexidade $O(n^2)$, possui um tempo de execução proporcional ao quadrado do tamanho da entrada n.

Por fim, as ordenações podem ser impactadas pela forma como o vetor dispõe-se antes de ser ordenado. Dessa forma, para um dado método de ordenação, diz-se que o **melhor caso** de ordenação ocorre quando há um menor custo de movimentações (trocas entre elementos) para que o vetor seja ordenado. Consequentemente, o custo de tempo, nesse caso, também será o menor possível. O **pior caso** é a situação em que existe a maior quantidade de trocas e maior gasto de tempo, e o **caso médio** é a situação intermediária entre os dois extremos mencionados. Uma vez apresentados conceitos básicos das ordenações, serão mostrados os comportamentos do Selection Sort e do Insertion Sort.

O Selection Sort é o método que percorre todo o vetor e seleciona, a cada iteração, o menor ou maior elemento da iteração (a depender de como deseja-se ordenar o vetor) e o coloca na posição correta do vetor, realizando as devidas trocas no final da iteração. Quando a iteração acaba, o valor recém reposicionado não será comparado novamente. A figura 3 ilustra o funcionamento do Selection Sort ordenando um vetor de forma ascendente da esquerda para a direita.

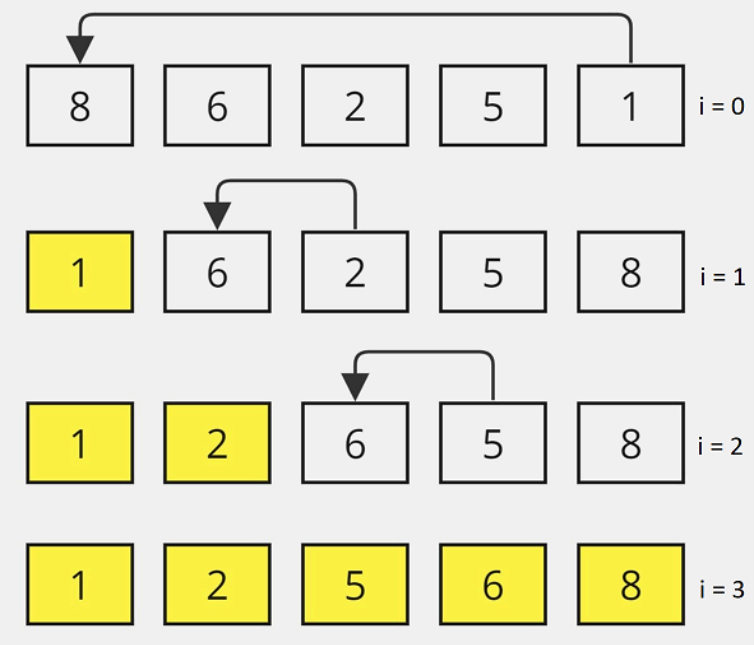


Figura 3 - Selection Sort

Desse modo, nota-se que o Selection Sort é um método de ordenação *in situ*, pois a ordenação é feita no próprio vetor, porém ele não é estável, já que o vetor é percorrido várias vezes, e o elemento a ser trocado é definido no momento da comparação. Esse movimento implica um cenário tal que, quando um elemento qualquer X possui valor igual a um outro elemento Y, os valores de ambos serão comparados na mesma iteração, entretanto, o elemento comparado por último é o elemento que será reposicionado, que normalmente estará localizado mais à direita desse vetor. Com respeito à ordem de complexidade do Selection Sort em relação ao espaço é O(1), já que não são criados subvetores. Em relação ao tempo, ele apresenta, para o pior caso, $O(n^2)$ comparações e O(n) trocas, logo, a ordem de complexidade é $O(n^2) + O(n) = O(n^2)$. Para o melhor caso, são $O(n^2)$ comparações e 0 trocas, portanto, a ordem de complexidade é $O(n^2) + 0 = O(n^2)$. Essas características fazem deste método um dos mais lentos tratando-se de tempo. Assim, este método é vantajoso por ter uma implementação fácil e por ser uma opção boa quando são poucos elementos a serem ordenados. Entretanto, quando o vetor já encontra-se ordenado previamente, o Selection ainda possui custo quadrático e apresenta um custo de tempo grande para uma quantidade maior de elementos.

Já o Insertion Sort possui uma forma de ordenação em que trocas sucessivas são realizadas em uma mesma iteração até que o elemento esteja posicionado adequadamente. Dessa forma, o algoritmo irá comparar o valor dos elementos no sentido da direita para a esquerda, e as trocas ocorrem até que o elemento em movimento encontre seu devido local no vetor, ordenando-o no sentido contrário (da esquerda para a direita). O vetor pode ser ordenado tanto ascendentemente quanto descendentemente. A figura 4 mostra a ordenação de um vetor seguindo o Insertion Sort.

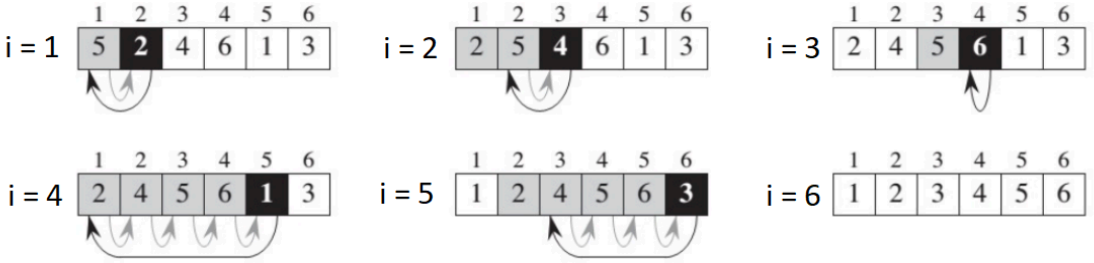


Figura 4 - Insertion Sort

Assim, é possível classificar o Insertion Sort como sendo um método estável, pois a ordem dos elementos de mesmo valor é mantida, já que a troca só ocorre se houver diferença entre os valores dos elementos, e ele é *in situ* porque o vetor original não utiliza subvetores adicionais fora do próprio vetor. Assim, por constituir um método *in situ*, o Insertion Sort apresenta ordem de complexidade com relação à espaço O(1). Com relação ao tempo, para o pior caso seriam $O(n^2)$ comparações e $O(n^2)$ trocas, que leva a uma ordem de $O(n^2) + O(n^2) = O(n^2)$. Já para melhor caso são O(n) comparações e 0 trocas, portanto, a ordem é $O(n) + 0 = O(n)$, ou seja, o custo é linear. Em suma, o Insertion Sort é um bom método quando os elementos estão quase ordenados, mas devido à sua grande quantidade de movimentações e custo quadrático para todos os casos que não são o melhor caso, assim como o Selection Sort, esse método não é tão eficiente quando o número de elementos a serem ordenados é grande.

É possível inferir, portanto, que não há como generalizar uma regra uma que estabeleça qual desses dois métodos é o melhor em qualquer caso. Para determinar qual método deve ser utilizado visando uma melhor otimização de uma ordenação, é necessário analisar em que cenário encaixa-se o conjunto dos elementos a serem ordenados para que seja determinado o algoritmo a ser implementado. Este raciocínio também é válido para quaisquer outros métodos de ordenação que não foram citados, uma vez que cada um possui pontos positivos e negativos. Assim, em teoria, o Insertion Sort é mais vantajoso quando o vetor já está quase ordenado e o Selection Sort é uma melhor opção quando o vetor está muito embaralhado.

Agora, serão mostradas as funções do simulador usadas na construção dos algoritmos. O simulador em questão é o MARS (MIPS Assembler and Runtime Simulator). Neste simulador, existem 32 registradores de 32 bits, que podem ser identificados pelo caracter "**$**" e possuem funções específicas. Além dos registradores, é possível interagir com a memória do programa, por meio de comandos que irão carregar o valor da memória em um registrador

ou salvar o valor de um registrador num espaço de memória. Dos registradores utilizados, tem-se:

$a0, $a1, $a2, $a3 - são registradores que servem como argumentos de funções.

$t1 a *$t9* - são registradores temporários

$s1 a *$s8* - são registradores similares aos temporários mas podem salvar valores

$v0 e *$v1* - armazenam valores retornados por funções. O registrador $v0 possui uma característica especial, de que quando usado com a função "li", que permite atribuir um valor arbitrário a qualquer registrador, é possível chamar a função "syscall", que executa uma função específica de acordo com o valor carregado por $v0.

Assim, tem-se as seguintes funções:

li $R1 CONST - atribui o valor de uma constante no registrador $R1

la $R1 ADDRESS - atribui um endereço de memória ao registrador $R1

move $R1 $R2 - atribui o valor do registrador $R2 ao registrador $R1

syscall - executa uma determinada operação, sendo que o código dessa operação é obtido do registrador $v0

LABEL: - permite criar um rótulo para um endereço de memória onde localiza-se uma determinada instrução ou dado, que pode possuir qualquer nome digitado pelo programador. Tal rótulo é o que permite a realização de importantes funções como laços de repetições e o estabelecimento de funções.

beq $R1 $R2 LABEL - compara o valor do registrador $R1 com o registrador $R2, se o valor do primeiro registrador é igual ao do segundo, salta para a linha onde encontra-se o rótulo digitado

ble $R1 $R2 LABEL - compara o valor do registrador $R1 com o registrador $R2, se o valor do primeiro registrador é menor ou igual ao do segundo, salta para a linha onde encontra-se o rótulo digitado

bge $R1 $R2 LABEL - compara o valor do registrador $R1 com o registrador $R2, se o valor do primeiro registrador é maior ou igual ao do segundo, salta para a linha onde encontra-se o rótulo digitado

bne $R1 $R2 LABEL - compara o valor do registrador $R1 com o registrador $R2, se o valor do primeiro registrador é diferente do segundo, salta para a linha onde encontra-se o rótulo digitado

j LABEL - realiza um salto para a linha onde encontra-se o rótulo digitado

addi $R1 $R2 CONST - realiza a soma de uma constante no valor do registrador $R2, armazenando o valor resultante no registrador $R1

subi $R1 $R2 CONST - realiza a subtração do valor do registrador $R2 por uma constante, armazenando o valor resultante no registrador $R1

sw $R1 LABEL($R2) - salva o valor contido no registrador $R1 dentro de um vetor definido pelo rótulo, no espaço de memória carregado pelo valor do registrador $R2

lw $R1 LABEL($R2) - carrega, no registrador $R1, o valor contido em um vetor definido pelo rótulo, no espaço de memória do valor do registrador $R2

III. Metodologia

Para a construção dos algoritmos, as funções apresentadas foram combinadas e separadas por módulos, cada qual realizando um procedimento necessário para a ordenação, e cada conjunto de funções foi rotulado para que fosse possível criar funções auxiliares e, também, para um melhor entendimento do código. Abaixo, um pseudocódigo do Selection Sort será mostrado como parâmetro para a elaboração do algoritmo em assembly:

```
1  Algorithm: SELECTIONSORT
   Input: int* v, int n
2  begin
3      for i ← 0 to i < n do
4          min ← i
5          for j ← i + 1 to j < n do
6              if v[j] < v[min] then
7                  min ← j
8              end
9          end
10         trocar v[i] com v[min]
11     end
12 end
```

Nos algoritmos, foram utilizadas duas diretivas que identificam as seções do programa em assembly. A seção **.data** é, normalmente, onde há a declaração de dados guardados na memória, que podem ser variáveis, strings ou vetores. Na implementação deste trabalho, essa seção possui a declaração do vetor com o rótulo "**vetor:**" usando o comando "**.alig 2**", e para esse comando, o número 2 indica que o vetor é de números inteiros. Em seguida, o foi utilizado o comando "**.space 40**", indicando que o vetor criado tem 40 bytes de tamanho, pois cada registrador consegue comportar até 4 bytes e o vetor possui 10 posições (4 * 10 = 40 bytes). Ainda nesta seção III do relatório será mostrado que foram feitos testes de ordenação com mais de 10 números. Para esses testes, bastou substituir o número de bytes consumidos de 40 por 80 na declaração do vetor e modificar alguns valores de registradores da função main. Além disso, ainda em .data tem mais um rótulo com nome "**msg_de_entrada:**" , seguido do comando "**.asciiz "Digite os 10 valores do vetor: "**". É por meio deste comando que a mensagem de saída aparece para o usuário, após ser carregada por um registrador do tipo $a. Em seguida, a diretiva **.text** define o início do código propriamente dito, assim sendo, é a partir daí que todas as instruções são passadas e executadas pelo processador. Dessa forma, todas as explicações que serão agora apresentadas referem-se ao conteúdo dessa parte.

Ambos os algoritmos começam pela entrada de dados, que são números não ordenados arbitrariamente escolhidos pelo usuário. Para isso, um loop é utilizado para a coleta desses dados, armazenados sequencialmente na memória. No momento da entrada de dados, o registrador $v0 é primeiramente carregado com o valor 4, que é o código para imprimir uma string (mensagem para a entrada de dados). Depois, o registrador $a0 carrega o endereço do rótulo que contém essa string e a função syscall é chamada para que a string apareça na tela do usuário. Dessa forma, o registrador $v0 é carregado com o valor 5, que representa entrada de dados e a função syscall é novamente chamada.

Todo loop utilizado no código possui o seguinte padrão: após o rótulo que define o loop, uma condicional de salto é feita para direcionar o fluxo de execução do algoritmo. Quando o resultado da condicional é verdadeiro, o fluxo segue para uma outra parte do código, e quando esse resultado é falso, o algoritmo vai para a próxima linha. Na última linha do loop, outro comando de salto deve ser chamado para que o algoritmo siga para o começo do loop novamente. Segue abaixo o trecho do código que mostra como as declarações são feitas:

```
entrada_de_dados: #rótulo do loop de declaração dos índices do vetor
    beq $t0 $t2 main #se i == 40(décima iteração) ir para main
    li $v0 5 #carregando o regsitrador $v0 com 5, que realiza a leitura de dados
    syscall #chamada da função de operação
    sw $v0 vetor($t0) #vet[i] = $v0 (número digitado pelo usuário)
    addi $t0 $t0 4 #i += 4
#a contagem acontece de 4 em 4 pois os endereços de memória são dispostos de 4 em 4
    j entrada_de_dados #reinicia o loop
```

No trecho, em azul, tem-se as operações, em vermelho, os registradores, em laranja, os comentários do código (que são sempre precedidos pelo caracter ‘#’), em lilás, declarações de rótulos e em preto tem-se as chamadas dos rótulos ou constantes inteiras definidas.

Depois que todos os números são coletados, os registradores do tipo $a são utilizados dentro da função de troca, que possui a finalidade de trocar as posições de dois elementos do vetor. Primeiramente, dois registradores ($a0 e $a1) são carregados com o endereço de dois contadores (i e j), e em seguida os espaços da memória são salvos de maneira cruzada, ou seja, se o valor carregado por $a0 vem do endereço de memória de i, então este valor deve ir para o endereço carregado por j. O mesmo processo é repetido para o registrador $a1.

Após a função de troca, inicia-se a função “main”, portanto, os registradores do tipo $t e $s são utilizados para armazenar valores que servirão como contadores, auxiliares ou então para carregar registradores com valores da memória, com a finalidade de que eles sejam comparados e, posteriormente, trocados de posição. Também foi implementado, em ambos os algoritmos, um mecanismo de cálculo do tempo de execução dos programas propositalmente instalado depois que a entrada é obtida, pois o tempo que o usuário leva para digitar os dados de entrada deve ser desconsiderado. Este mecanismo consiste em marcar a hora atual de quando o programa passa por uma determinada linha, e obter o tempo decorrido do programa realizando a subtração da marca final pela marca inicial de tempo. O objetivo desse cálculo é verificar quanto tempo levam as ordenações em cada caso de teste.

Dentro da função main, dois laços de repetição são estabelecidos, como mostrado no pseudocódigo. O laço externo é o que determina quantas vezes o vetor é percorrido, e a cada iteração, os índices do vetor são lidos, para que ao final desse processo o menor valor seja encontrado. Além disso, toda vez que o laço externo começa, a posição do menor valor é atualizada pois, caso contrário, o menor valor do vetor seria encontrado em todas as iterações, e o algoritmo entraria numa repetição em que apenas o menor valor do vetor é colocado no começo. O laço interno determina quais elementos serão comparados, e quando um valor menor é encontrado, a posição desse elemento é guardada por um registrador, para que ao final do laço interno haja a troca desse elemento na posição correta. Por fim, quando o loop externo termina, o algoritmo deve encerrar sua execução. A figura 5 mostra um fluxograma que representa a execução desse algoritmo.

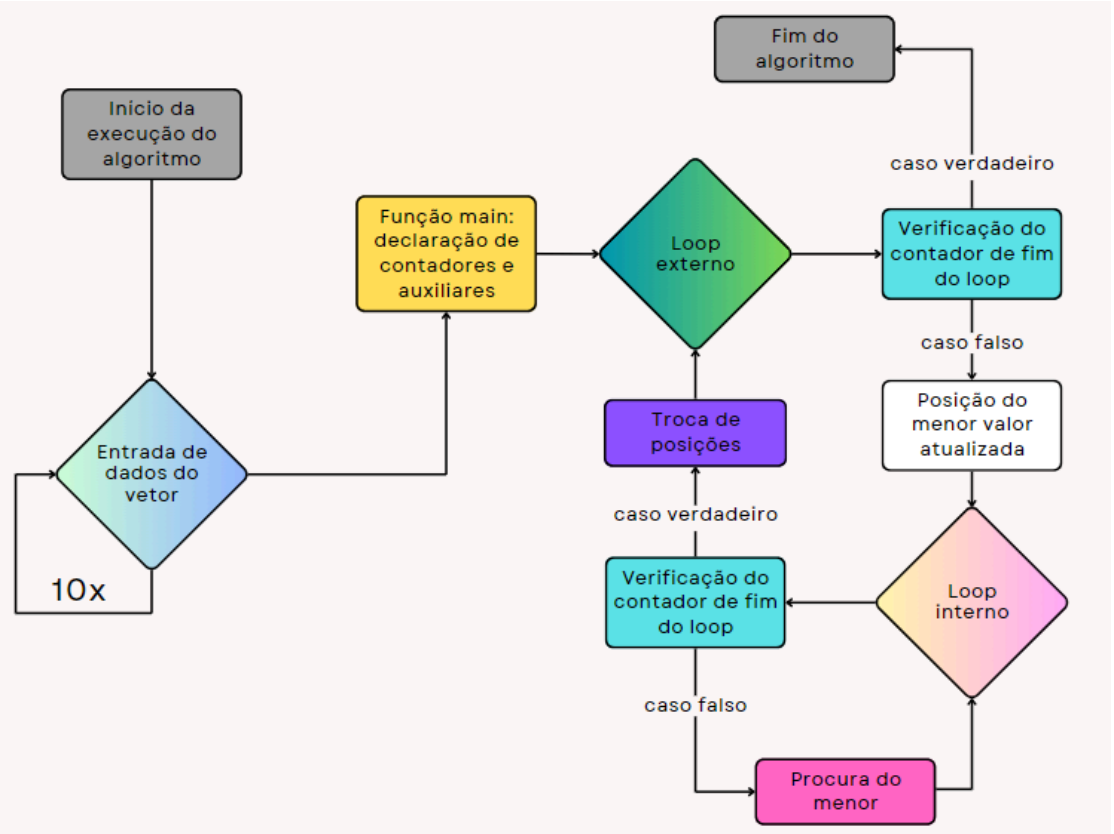


Figura 5 - fluxograma do algoritmo do Selection Sort

O algoritmo do Insertion Sort inicia de maneira similar ao último, com os mesmos comandos para a entrada dos valores do vetor e a declaração de variáveis na função main. No entanto, o loop interno deste algoritmo consiste nas trocas sucessivas mencionadas, que continuam sendo efetuadas enquanto o elemento da iteração não for colocado na posição correta. Isto pode ser observado no pseudocódigo abaixo:

```
1  Algorithm: INSERTIONSORT
   Input: int* v, int n
2  begin
3      for i ← 1 to i < n do
4          aux ← v[i]
5          j ← i − 1
6          while j >= 0  &  aux < v[j] do
7              v[j + 1] ← v[j]
8              j ← j − 1
9          end
10         v[j + 1] ← aux
11     end
12 end
```

Nele, vê-se que a condição de parada do loop em questão é que o contador do loop interno (j) tenha chegado ao início do vetor (posição 0) ou que o elemento comparado seja menor do que o elemento que está sendo trocado sucessivamente, indicando que este já encontrou a sua posição relativa no vetor.

Dessa forma, analogamente ao pseudocódigo, o algoritmo conta com um registrador que armazena o valor da auxiliar (também chamada de key ou chave), que diferentemente da auxiliar do último algoritmo, esta armazena um valor do vetor, ao invés de uma posição dele. Ao passo que o loop interno (while) inicia, o valor da chave e do contador interno são atualizados, para que iniciem-se as trocas. Assim, os valores vão sendo continuamente atualizados enquanto j decresce. Por fim, quando o laço interno termina, o vetor realiza um último movimento para restaurar o valor que foi sobrescrito na primeira troca de valores. A figura 6 esquematiza todos os processos descritos do algoritmo do Insertion Sort por um fluxograma.

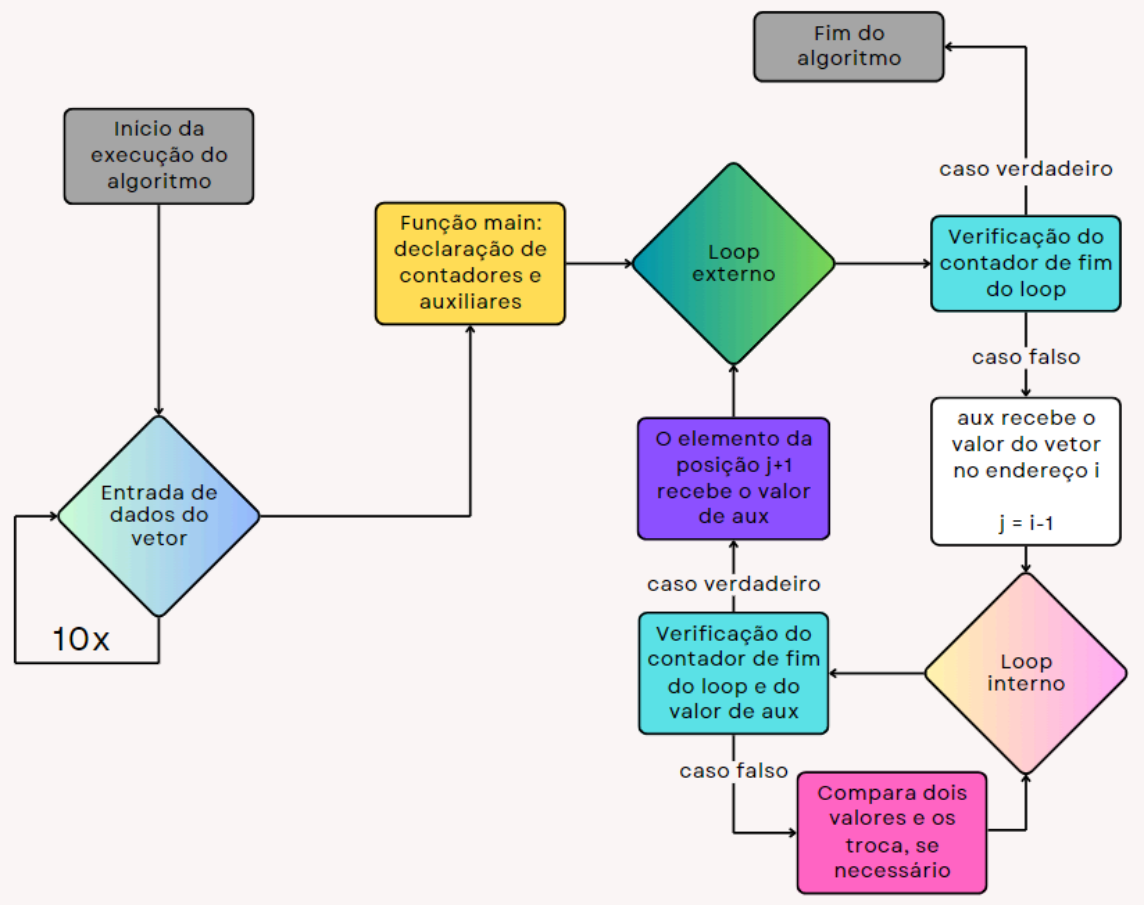


Figura 6 - fluxograma do algoritmo do Insertion Sort

Os algoritmos em análise foram testados quanto à ordenação correta e sequencial de todos os números na memória, além do teste do tempo de execução de ambos, com o intuito de que os métodos sejam comparados posteriormente. Para isso, foram realizados diferentes testes, todos eles possuem uma entrada de dados com números desordenados. Intencionalmente, os testes 1 e 3 tendem a favorecer a ordenação do Selection Sort, já que os vetores estão no pior caso possível para o Insertion Sort, assim como os testes 2 e 4 tendem a favorecer o Insertion Sort em detrimento do Selection Sort, pois os números estão quase em suas posições corretas. Como os testes 3 e 4 possuem um número maior de números a serem ordenados, algumas modificações no algoritmo precisaram ser realizadas para que o vetor conseguisse comportar mais posições (como já mencionado), no entanto, os trechos do código relativos à ordenação permanecem os mesmos. O motivo de esses dois testes possuírem mais números é uma tentativa de acentuar a diferença entre o tempo de execução dos dois algoritmos, a fim de comprovar que as condições prévias do vetor fazem um método ser menos custoso que outro numa dada situação. Por fim, o teste 5 deve comprovar que os algoritmos conseguem ordenar elementos com números maiores que os demais testes.

Assim, as entradas dos testes são:

Teste 1: 10 9 8 7 6 5 4 3 2 1
Teste 2: 1 3 2 5 4 8 6 7 9 10
Teste 3: 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Teste 4: 4 2 1 3 8 10 9 5 7 11 6 13 14 20 12 17 16 19 18 15
Teste 5: 849 665 775 122 316 1847 159 375 360 1318

Os resultados esperados da ordenação para cada caso de teste são:

Testes 1 e 2: 1 2 3 4 5 6 7 8 9 10
Testes 3 e 4: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Teste 5: 122 159 316 360 375 665 775 849 1318 1847

IV. Resultados

Para os 5 testes, tanto o Selection Sort quanto o Insertion Sort demonstraram taxas de acerto de 100%, portanto, em todos os casos, os algoritmos ordenaram corretamente os números de entrada. As figuras de 7.1 a 11.2 mostram as entradas dos números no simulador MARS, na caixa de input e também na memória. A figura 12 mostra o resultado da ordenação em ambos os algoritmos para os dois primeiros testes e as figuras 13 e 14 mostram os resultados das ordenações dos testes 4 e 5, respectivamente, em ambos os algoritmos.

O registrador $s5 foi utilizado para armazenar a subtração das marcas de tempo dos códigos (hora final - hora inicial). A figura 15 mostra o resultado, em relação ao tempo (dado em milissegundos) dos testes 1, 2, 5 para o Selection Sort e teste 4 para o Insertion Sort, com 2 milissegundos. A figura 16 mostra o tempo decorrido nos testes 3 e 4 para o Selection Sort, com 7 milissegundos. A figura 17 mostra o tempo decorrido para o teste 1 do Insertion Sort no teste 1, com 3 milissegundos. A figura 18 mostra o tempo decorrido nos testes 2 e 5 do Insertion Sort, com 1 milissegundo e, por fim, a figura 19 mostra o tempo decorrido no teste 3 do Insertion Sort, com 9 milissegundos.

Para uma melhor visualização desses resultados, a tabela abaixo mostra os resultados dos tempos:

	Teste 1	**Teste 2**	**Teste 3**	**Teste 4**	**Teste 5**
Selection Sort	2ms	2ms	7ms	7ms	2ms
Insertion Sort	3ms	1ms	9ms	2ms	1ms

Vale ressaltar que, nos testes 1, 2 e 5, os algoritmos foram projetados para armazenar as 10 primeiras posições da memória. Por isso, o vetor manipulado pode ser observado na primeira linha e nas duas primeiras colunas da segunda linha. Para os testes 3 e 4, o vetor pode ser observado nas duas primeiras linhas e na quarta coluna da terceira linha. Nas figuras dos resultados, a tabela representa a memória do simulador, por isso, é possível notar que os endereços de memória são saltados de 4 em 4 (ao lado de “Value” acima da primeira linha), isso deve-se ao fato de que cada endereço consegue armazenar até 4 bytes, que é também o motivo pelo qual, nos algoritmos criados, os contadores aumentam de valor de 4 em 4 a cada iteração, ao invés de aumentarem de 1 em 1. Dessa forma, é mais fácil realizar as devidas operações que demandam o endereçamento dos índices do vetor.

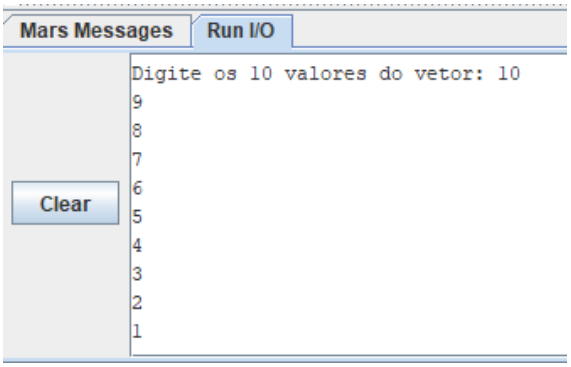
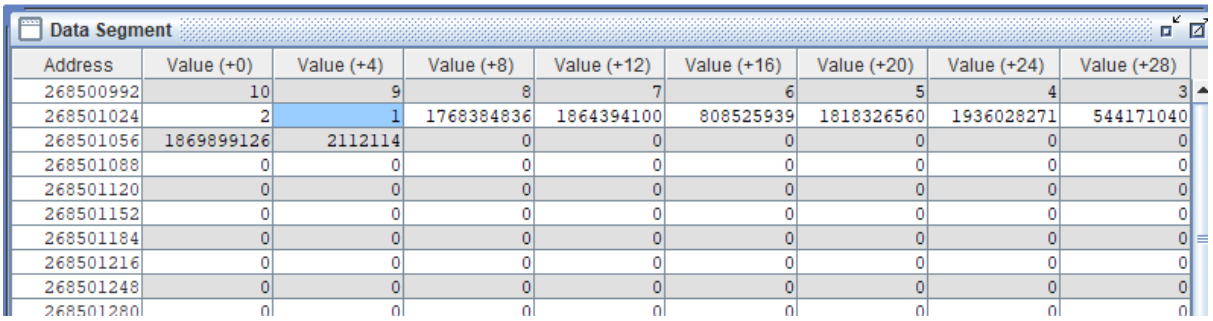


Figura 7.1 - entrada do teste 1 - input do simulador



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	10	9	8	7	6	5	4	3
268501024	2	1	1768384836	1864394100	808525939	1818326560	1936028271	544171040
268501056	1869899126	2112114	0	0	0	0	0	0
268501088	0	0	0	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0

Figura 7.2 - entrada do teste 1 - memória do simulador

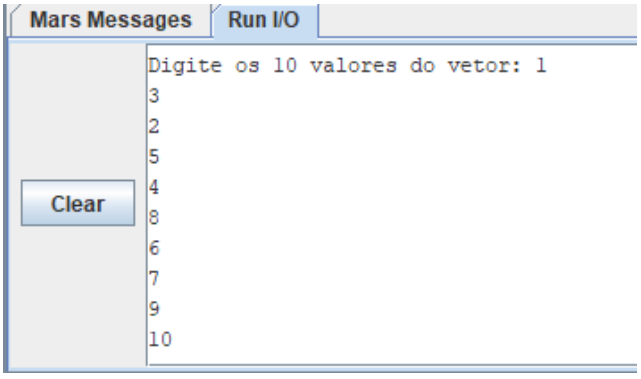
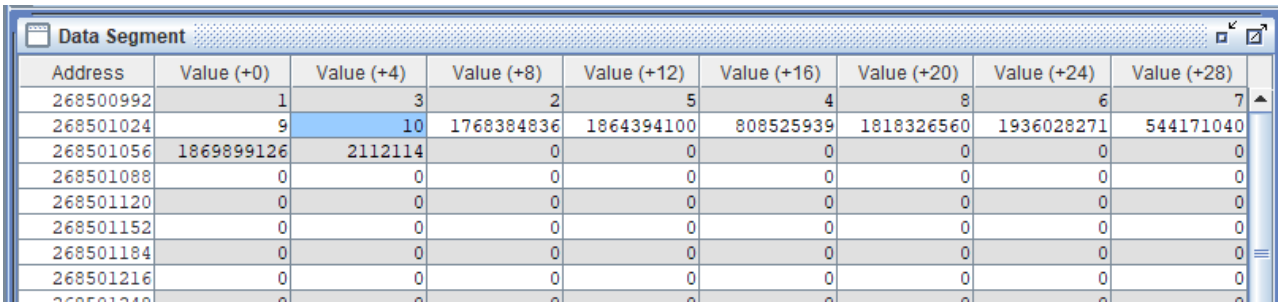


Figura 8.1 - entrada do teste 2 - input do simulador



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	1	3	2	5	4	8	6	7
268501024	9	10	1768384836	1864394100	808525939	1818326560	1936028271	544171040
268501056	1869899126	2112114	0	0	0	0	0	0
268501088	0	0	0	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0

Figura 8.2 - entrada do teste 2 - memória do simulador

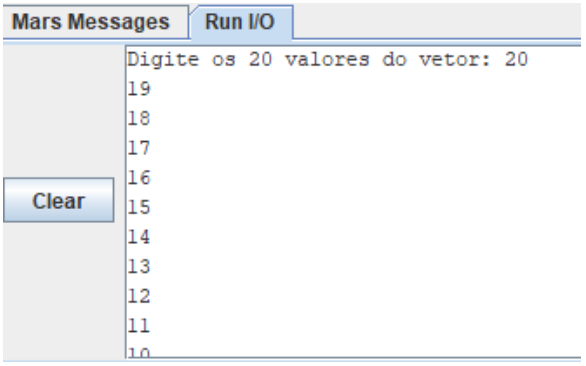


Figura 9.1 - entrada do teste 3 - input do simulador, parte 1

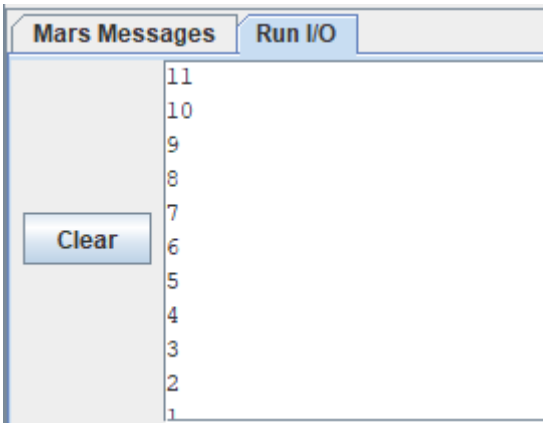


Figura 9.2 - entrada do teste 3 - input do simulador, parte 2

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	20	19	18	17	16	15	14	13
268501024	12	11	10	9	8	7	6	5
268501056	4	3	2	1	1768384836	1864394100	808591475	1818326560
268501088	1936028271	544171040	1869899126	2112114	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0
268501312	0	0	0	0	0	0	0	0

Figura 9.3 - entrada do teste 3 - memória do simulador

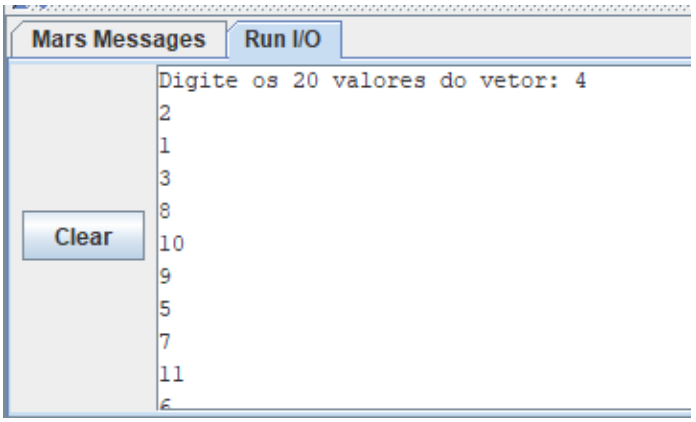


Figura 10.1 - entrada do teste 4 - input do simulador, parte 1

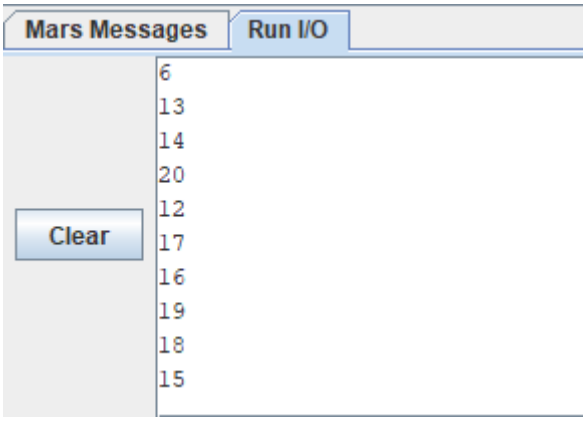


Figura 10.2 - entrada do teste 4 - input do simulador, parte 2

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	4	2	1	3	8	10	9	5
268501024	7	11	6	13	14	20	12	17
268501056	16	19	18	15	1768384836	1864394100	808591475	1818326560
268501088	1936028271	544171040	1869899126	2112114	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0

Figura 10.3 - entrada do teste 4 - memória do simulador

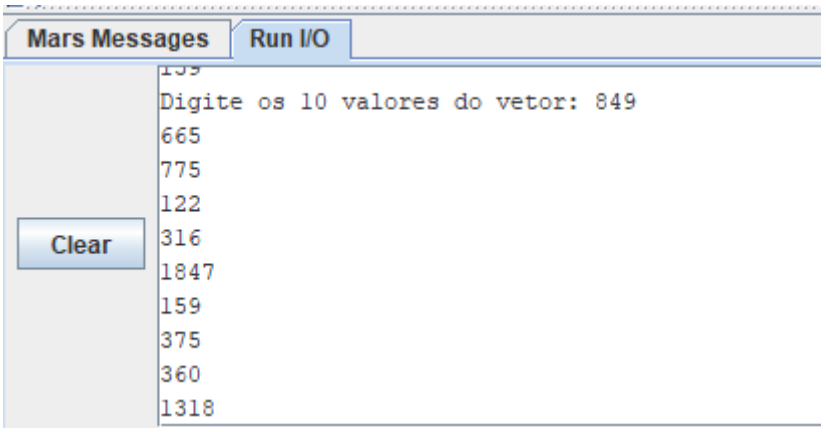


Figura 11.1 - entrada do teste 5 - input do simulador

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	849	665	775	122	316	1847	159	375
268501024	360	1318	1768384836	1864394100	808525939	1818326560	1936028271	544171040
268501056	1869899126	2112114	0	0	0	0	0	0
268501088	0	0	0	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0

Figura 11.2 - entrada do teste 5 - memória do simulador

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	1	2	3	4	5	6	7	8
268501024	9	10	1768384836	1864394100	808525939	1818326560	1936028271	544171040
268501056	1869899126	2112114	0	0	0	0	0	0
268501088	0	0	0	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0

Figura 12 - resultado dos testes 1 e 2 - memória do programa

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	1	2	3	4	5	6	7	8
268501024	9	10	11	12	13	14	15	16
268501056	17	18	19	20	1768384836	1864394100	808591475	1818326560
268501088	1936028271	544171040	1869899126	2112114	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0

Figura 13 - resultado dos testes 3 e 4 - memória do programa

gment

	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
2	122	159	316	360	375	665	775	849
4	1318	1847	1768384836	1864394100	808525939	1818326560	1936028271	544171040
6	1869899126	2112114	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Figura 14 - resultado do teste 5 - memória do programa

$s3	19	0
$s4	20	0
$s5	21	2
$s6	22	0
$s7	23	0

Figura 15 - resultados - registrador de tempo com 2ms

$s4	20	0
$s5	21	7
$s6	22	0
$s7	23	0

Figura 16 - resultados - registrador de tempo com 9ms

$s3	19	0
$s4	20	0
$s5	21	3
$s6	22	0
$s7	23	0
$t8	24	0

Figura 17 - resultados - registrador de tempo com 3ms

$s4	20	0
$s5	21	1
$s6	22	0
$s7	23	0
$t8	24	0

Figura 18 - resultados - registrador de tempo com 1ms

$s3	19	0
$s4	20	0
$s5	21	9
$s6	22	0
$s7	23	0
$t8	24	0

Figura 19 - resultados - registrador de tempo com 9ms

A seguir, a figura 20 mostra o gráfico que contém os resultados, em percentual, da precisão de acerto dos algoritmos. Nele, a precisão de acerto do Selection Sort está em vermelho, e a precisão de acerto do Insertion Sort, em azul escuro.

Figura 20 - gráfico avaliativo da taxa de acerto dos algoritmos

A figura 21 traz outro gráfico, dessa vez comparando os tempos de execução do Insertion Sort e do Selection Sort em cada caso de teste, seguindo o mesmo padrão de cores:

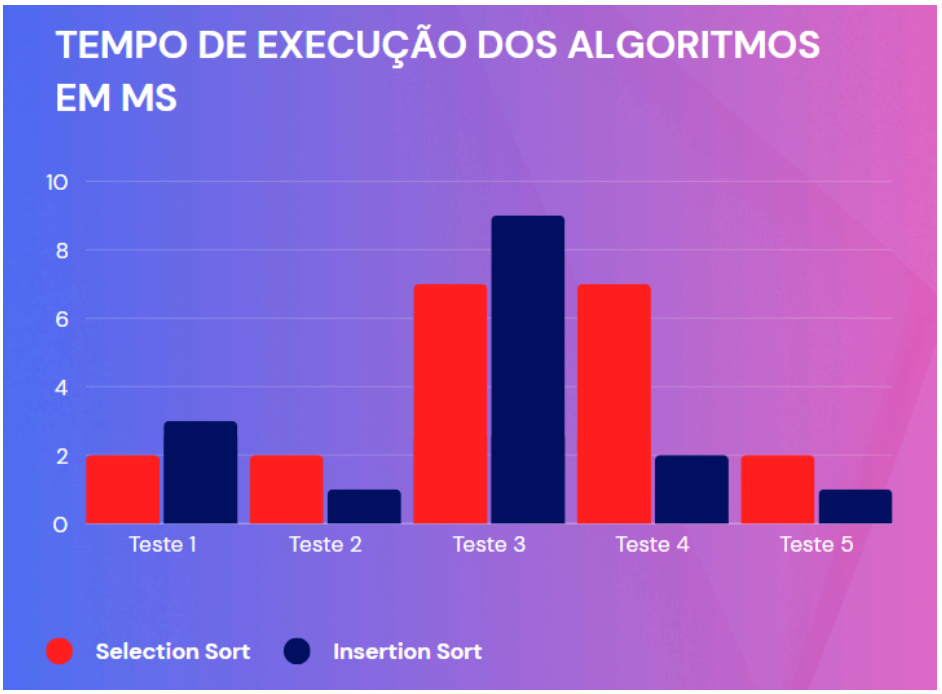


Figura 21 - gráfico avaliativo do tempo de execução dos algoritmos

Com os resultados em mãos, é possível observar que, nos testes 1 a 4, o condicionamento prévio da ordem dos elementos fez diferença quanto ao tempo de execução dos algoritmos. Em síntese, comprova-se que, de fato, quando os elementos estão muito embaralhados, o Selection Sort tende a ser mais rápido, como ocorreu nos testes 1 e 3. Por outro lado, nos testes 2 e 4, em que os números estavam quase ordenados, o Insertion Sort performou uma ordenação muito mais ágil. Observa-se, ainda, que os tempos de execução do Selection Sort foram exatamente iguais em todos os testes com o mesmo número de elementos de entrada, como nos testes 1, 2 e 5 (10 elementos) e nos testes 3 e 4 (20 elementos), evidenciando que o Selection Sort depende exclusivamente do número de elementos a serem ordenados. Já o Insertion Sort possui uma variação de tempo muito maior de caso para caso, portanto, a ordenação por este método é mais otimizada quanto mais perto da ordem crescente o vetor se encontra, pois menos movimentos são feitos e mais etapas de troca são ignoradas.

V. Conclusão

Este trabalho possui como finalidade a elaboração de dois algoritmos de ordenação na linguagem de programação assembly, com o auxílio do simulador MARS que representa uma arquitetura MIPS. O primeiro algoritmo é o Selection Sort e o segundo algoritmo é o Insertion Sort.

O simulador pode executar diversas funções diferentes, incluindo: carregar valores nos registradores, realizar operações matemáticas, condicionar o direcionamento do fluxo de execução do programa por meio de comparações com saltos, interagir com a memória e imprimir números e palavras na tela do usuário. Assim, os algoritmos seguem um fluxo parecido, mas diferem quanto ao número de movimentos e ao tempo de execução. As funções foram utilizadas de forma específica para que conseguissem reproduzir o comportamento dos dois algoritmos de ordenação. O Selection Sort é o método que irá selecionar qual o menor valor do vetor a cada iteração e o posicionará no vetor da esquerda para a direita, ascendentemente. Por outro lado, o Insertion Sort é o método que realiza trocas sucessivas da direita para a esquerda, até que o número em movimento chegue ao espaço correto do vetor ou quando o limite inicial do vetor é atingido. Os métodos possuem ordem de complexidade quadrática na maioria dos casos e são vantajosos quando o número de elementos a serem ordenados é pequeno. À respeito da metodologia de avaliação dos resultados, os algoritmos foram testados quanto ao desempenho na ordenação sequencial crescente dentro da memória do simulador e quanto ao tempo gasto em cada teste e por cada método. Logo, espera-se que os endereços iniciais da memória do simulador possuam os dados da entrada de forma ordenada.

Dos resultados, ambos os algoritmos tiveram sucesso em todos os testes a que foram submetidos, uma vez que os dados de entrada encontram-se nos locais corretos e estão dispostos de forma crescente. Além disso, foi constatado que o Selection Sort é um método de ordenação mais rápido que o Insertion Sort quando o vetor está muito embaralhado, enquanto o Insertion Sort provou um tempo de ordenação menor quando o vetor está próximo da ordenação.

Concluindo, os algoritmos representam corretamente o comportamento de ambos os métodos de ordenação, com taxas de acerto de 100% em todos os casos testados, e o condicionamento prévio dos elementos que seriam ordenados provou ter implicações no tempo em que cada método leva para ordenar todos os números.