# Relatório - Laboratório 1

**Eduardo Ferreira de Assis, 17/0102289**
**Thiaggo Ferreira Bispo de Souza, 17/0157024**
**Emanoel Johannes Cardim Lazaro. 17/0140997**
**Alexandre Souza Costa Oliveira, 17/0098168**
**Gabriel Pinheiro dos Santos, 170103579**
**Maurílio de Jesus Silveira , 17/0152294**

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CiC 116394 - Organização e Arquitetura de Computadores - Turma A

## Questões

1.3. Analisando o código *sort.s*, modelamos 2 equações para representar a quantidade de instruções ($I$) utilizadas:

1.3.1. Equação para o melhor caso:

$$I_0 = 9 + 5 + (n-1) * (2 + 6 + 2) + 1 + 7$$

Sendo $n$ o tamanho do vetor. Observamos que, no melhor caso, o algoritmo só irá fazer a checagem dos valores, sem realizar nenhuma troca. Logo, ele percorrerá $V[] = \{0, 1, ..., n-1\}$.

1.3.2. Equação para o pior caso:

$$I_i = 9 + 5 + (x-1) * (2 + (11+7) * \frac{x}{2} + 3) + 1 + 7$$

Sendo $x$ o tamanho do vetor. Observamos que, para o pior caso, o algoritmo irá realizar $x-1$ checagens, e para cada iteração realizará $\frac{x}{2}$ trocas (a função representa o número de trocas médias, i.e., $\frac{x*(x-1)}{2*(x-1)}$)

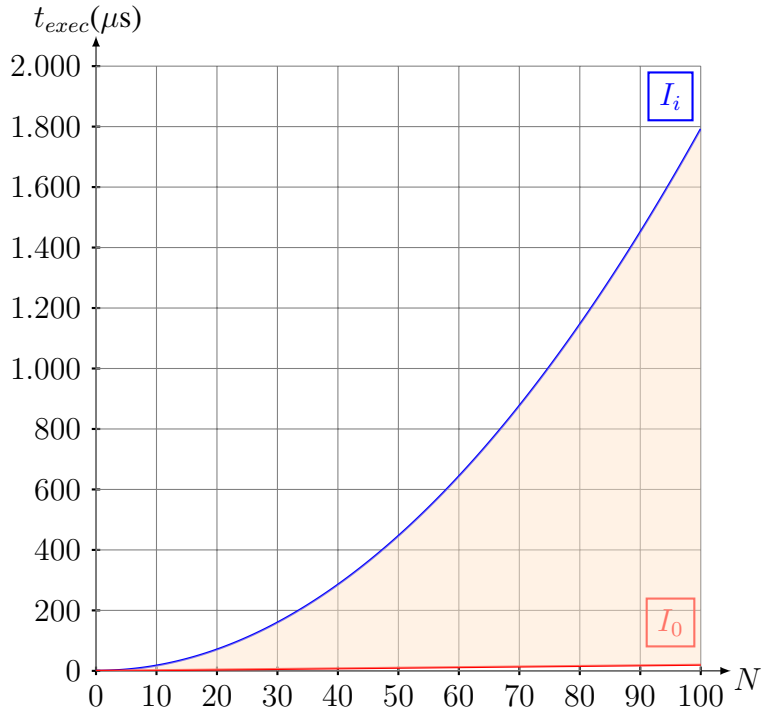


Gráfico comparativo do tempo de execução para $I_0$, $I_i$.

**Tabela 1. Tabela Melhor e Pior Caso**

| Entrada | Saídas | | | |
|---|---|---|---|---|
| $n$ | $I_0$ | $t_{exec}I_0$ | $I_i$ | $t_{exec}I_i$ |
| **10** | 112 | 2,24 | 877 | 17,54 |
| **20** | 212 | 4,24 | 3537 | 70,74 |
| **30** | 312 | 6,24 | 7997 | 159,94 |
| **40** | 412 | 8,24 | 14257 | 285,14 |
| **50** | 512 | 10,24 | 22317 | 446,34 |
| **60** | 612 | 12,24 | 32177 | 643,54 |
| **70** | 712 | 14,24 | 43837 | 876,74 |
| **80** | 812 | 16,24 | 57297 | 1145,94 |
| **90** | 912 | 18,24 | 72557 | 1451,14 |
| **100** | 1012 | 20,24 | 89617 | 1792,34 |

1.4.

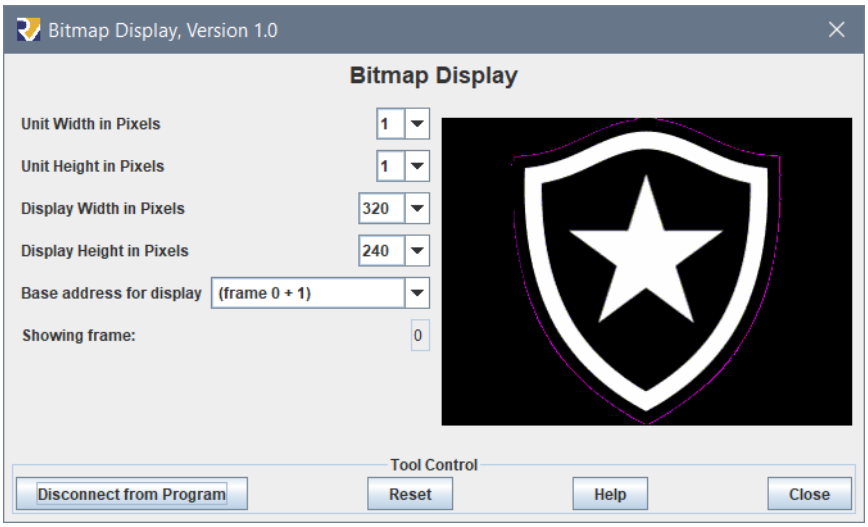


**Figura 1. Bandeira do Botafogo printada no beatmap display**

2.2. Foi usado o site `https://cx.rv8.io/` como cross compiler. As alterações foram feitas de modo a executar o sortc.s no RARS

```
.data           #adicionado
v:
  .word 9
  .word 2
  .word 5
  .word 1
  .word 8
  .word 2
  .word 4
  .word 3
          .word 6
  .word 7
.LC0:
  .string "%d\t"


.text           #adicionado

jal zero,main
show:           #modificado de show(int*, int) para show
  addi sp,sp,-48
  sw ra,44(sp)
  sw s0,40(sp)
  addi s0,sp,48
  sw a0,-36(s0)
  sw a1,-40(s0)
  sw zero,-20(s0)
.L3:
  lw a4,-20(s0)
  lw a5,-40(s0)
  bge a4,a5,.L2
  lw a5,-20(s0)
  slli a5,a5,2
  lw a4,-36(s0)
  add a5,a4,a5
  lw a5,0(a5)
```

**Figura 2. Alterações feitas no código**

```
  mv a0,a5                    #agora move o a5 para o a0, ou seja, coloca o número a ser printado no a0
  #lui a5,%hi(.LC0)           removido pois coloca uma string no a0 e não um número
  #addi a0,a5,%lo(.LC0)       removido pois coloca uma string no a0 e não um número
  #call printf                trocar o printf por uma rotina que faça o mesmo trabalho (representada nas linhas comentadas logo abaixo)
  addi a7,zero, 1       # colocar o 1 no a7
  ecall                 #print int
  addi a7,zero,11       #a7 = 11
  addi a0,zero,'\t'     #a0 = '\t'
  ecall                 #print char
  lw a5,-20(s0)
  addi a5,a5,1
  sw a5,-20(s0)
  j .L3
.L2:
  li a0,10
  #call putchar          fazer rotina semelhante ao putchar (representada nas linhas comentadas logo abaixo)
  addi a7, zero, 11     # a7 recebe 11
  mv t0,a0              # coloca a0 em t0
  addi a0, zero, '\n'   # a0 = "\n"
  ecall                 # print char
  mv a0,t0              # coloca t0 em a0
  nop
  lw ra,44(sp)
  lw s0,40(sp)
  addi sp,sp,48
  jr ra
swap:           #modificado de swap(int*, int) para swap
  addi sp,sp,-48
  sw s0,44(sp)
  addi s0,sp,48
  sw a0,-36(s0)
  sw a1,-40(s0)
  lw a5,-40(s0)
  slli a5,a5,2
  lw a4,-36(s0)
  add a5,a4,a5
```

**Figura 3. Alterações feitas no código**

```
  lw a5,0(a5)
  sw a5,-20(s0)
  lw a5,-40(s0)
  addi a5,a5,1
  slli a5,a5,2
  lw a4,-36(s0)
  add a4,a4,a5
  lw a5,-40(s0)
  slli a5,a5,2
  lw a3,-36(s0)
  add a5,a3,a5
  lw a4,0(a4)
  sw a4,0(a5)
  lw a5,-40(s0)
  addi a5,a5,1
  slli a5,a5,2
  lw a4,-36(s0)
  add a5,a4,a5
  lw a4,-20(s0)
  sw a4,0(a5)
  nop
  lw s0,44(sp)
  addi sp,sp,48
  jr ra
sort:           #modificado de sort(int*, int) para sort
  addi sp,sp,-48
  sw ra,44(sp)
  sw s0,40(sp)
  addi s0,sp,48
  sw a0,-36(s0)
  sw a1,-40(s0)
  sw zero,-20(s0)
.L9:
  lw a4,-20(s0)
  lw a5,-40(s0)
  bge a4,a5,.L10
```

**Figura 4. Alterações feitas no código**

```
109   lw a5,-20(s0)
110   addi a5,a5,-1
111   sw a5,-24(s0)
112 .L8:
113   lw a5,-24(s0)
114   bltz a5,.L7
115   lw a5,-24(s0)
116   slli a5,a5,2
117   lw a4,-36(s0)
118   add a5,a4,a5
119   lw a4,0(a5)
120   lw a5,-24(s0)
121   addi a5,a5,1
122   slli a5,a5,2
123   lw a3,-36(s0)
124   add a5,a3,a5
125   lw a5,0(a5)
126   ble a4,a5,.L7
127   lw a1,-24(s0)
128   lw a0,-36(s0)
129   call swap     #modificado de swap(int*, int) para swap
130   lw a5,-24(s0)
131   addi a5,a5,-1
132   sw a5,-24(s0)
133   j .L8
134 .L7:
135   lw a5,-20(s0)
136   addi a5,a5,1
137   sw a5,-20(s0)
138   j .L9
139 .L10:
140   nop
141   lw ra,44(sp)
142   lw s0,40(sp)
143   addi sp,sp,48
144   jr ra
```

**Figura 5. Alterações feitas no código**

```
134 .L7:
135   lw a5,-20(s0)
136   addi a5,a5,1
137   sw a5,-20(s0)
138   j .L9
139 .L10:
140   nop
141   lw ra,44(sp)
142   lw s0,40(sp)
143   addi sp,sp,48
144   jr ra
145 main:
146   addi sp,sp,-16
147   sw ra,12(sp)
148   sw s0,8(sp)
149   addi s0,sp,16
150   li a1,10
151   lui a5,%hi(v)
152   addi a0,a5,%lo(v)
153   call show     #modificado de show(int*, int) para show
154   li a1,10
155   lui a5,%hi(v)
156   addi a0,a5,%lo(v)
157   call sort
158   li a1,10
159   lui a5,%hi(v)
160   addi a0,a5,%lo(v)
161   call show
162   li a5,0
163   mv a0,a5
164   lw ra,12(sp)
165   lw s0,8(sp)
166   addi sp,sp,16
167   #jr ra               removido ra do final, pois é final do programa e adicionadas as linhas de código abaixo
168   addi a7, zero,10    #a7 = 10
169   ecall               #exit
```

**Figura 6. Alterações feitas no código**

2.3.

| | Número total de instruções / Tamanho do código em bytes | | | | |
|---|---|---|---|---|---|
| **Programa** | **-O0** | **-O1** | **-O2** | **-O3** | **-Os** |
| **sortc.s** | 1748 / 560 | 713 / 432 | 554 / 372 | 494 / 356 | 831 / 420 |
| **sortc2.s** | 1807 / 576 | 726 / 432 | 525 / 360 | 506 / 356 | 807 / 404 |

**Figura 7. Alterações feitas no código**

A análise de -O0 a -Os mostra que o programa sortc.s possui, no estado mais baixo de otimização, um número de instruções menor e ocupa menos espaço na memória que o programa sortc2.s, porém, com o aumento da otimização, o programa sortc.s e o programa sortc2.s têm seus números de instruções e tamanhos reduzidos até a compilação com nível de otimização -Os, em que há um incremento no número de instruções e no tamanho de ambos os códigos. No entanto, o aumento no tamanho em bytes pode ser explicado pelas linhas de código adicionados aos programas (funções printf, putchar, etc) para que funcionem no RARS, uma vez que, de acordo com o site `https://bit.ly/100nop0` a opção de otimização -Os tem como foco o aprimoramento em relação ao tamanho do código.

Além disso, no item 1.1 foi calculado que o código de máquina do programa sort.s ocupa 280 bytes de memória e executa 732 instruções, o que, comparado ao número de instruções e ao tamanho dos programas

comparados na tabela no nível de optimização -O1, nos mostra que é possível obter um código de mesma função com um número de instruções instruções um pouco maior, mas com menor tamanho em bytes.

3.4. Link do vídeo da questão `https://www.youtube.com/watch?v=it0YoaeJbvI`

3.5.

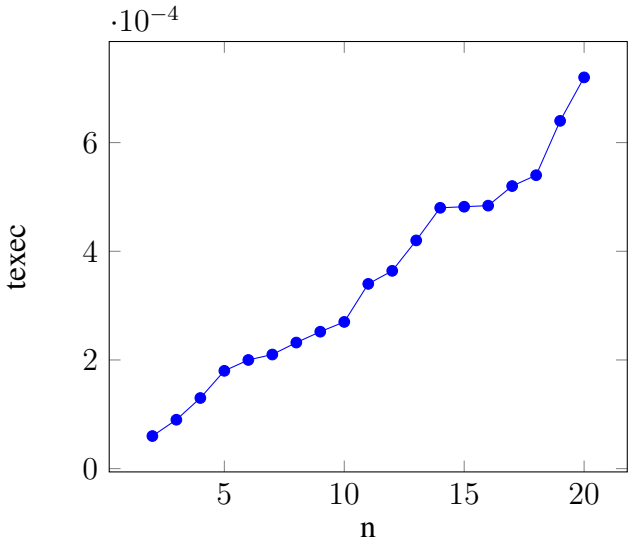


**Gráfico representando o tempo de execução (*texec*) em segundos a $10^4$ para *n* entradas (número de instruções).**

Analisando o gráfico acima, percebemos que o algoritmo de ORDENAÇÃO tem complexidade linear, sendo as chamadas de *ecall* para desenhar a linha entre os vértices o fator de maior variância no número de instruções.

Vale ressaltar que o algoritmo desenvolvido é diferente do demonstrado na ementa do laboratório. O algoritmo passado pelo professor envolve programação dinâmica e permutações, por ele, devemos testar todos os caminhos possíveis para concluir o menor caminho. Logo, vemos que o algoritmo tem complexidade exponencial (Caixeiro viajante).

A solução aplicada pelo grupo acha uma aproximação do menor caminho, de maneira direta (nem sempre é o menor caminho). O algoritmo procura o menor caminho a partir de tal vértice, fazendo isso *n* vezes. Dessa forma, encontramos o menor caminho entre os vizinhos em uma checagem da matriz de distâncias.

Para efeitos de comparação, usando o algoritmo do Caixeiro Viajante, o número de iterações para $n = 100$ seria por volta de $4,019693683 * 10^14$, no algoritmo desenvolvido, para $n = 100$, o número seria de $100$ iterações (considerando somente a lógica da construção do menor caminho). Ou seja, para *n* muito grande a solução pelo algoritmo proposto seria inviável computacionalmente, levando algumas centenas de trilhões de anos para obter o resultado, já por outro lado, utilizando o nosso algoritmo, seria praticamente instantâneo.