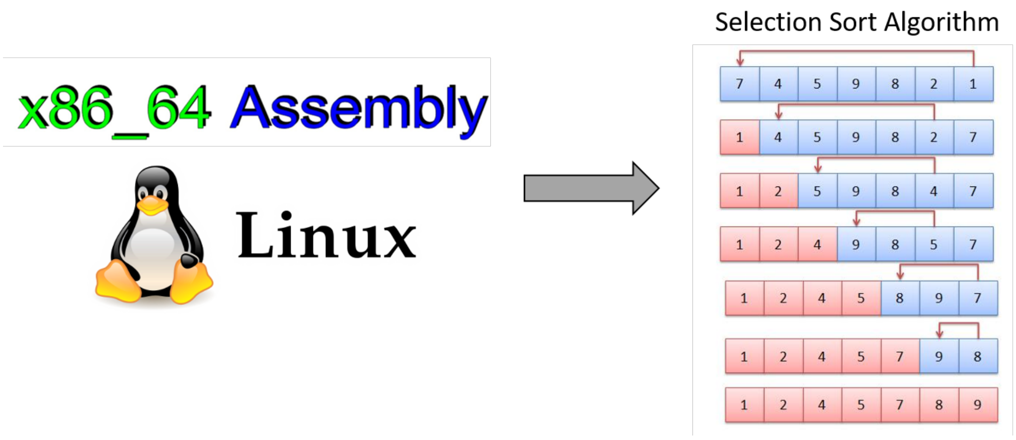
Assignment 1 - Sorting in Assembler

Henrik Bastholm, Frank Frederiksen-Møller & Kristian Høybye
hebas16, frfre16 & krhoe16
Supervisor: Richard Röttger
DM548 - Computer Architecture

November 12, 2017



Contents

1 The algorithm

The chosen sorting algorithm to be implemented in Assembly 86_64 was selection sort.
Selection sort is a simple sorting algorithm that is relatively easy implemented so it was possible to mainly focus on the use of the Assembly language. This means the algorithm isn't going to be the most efficient when it comes to runtime, but it gave the opportunity to focus more on the actual Assembly language, and getting the code to work, which was decided to be more important.
The program will first allocate memory according to the given file's file-size. It will then read the file and store the context in memory as ASCII values. Another buffer is then allocated in memory by the amount of numbers multiplied by 8(since each number allocates 8 bytes). Then the ASCII values is rewritten into integers and stored in the second buffer.

Since the integers are stored in a buffer, it gives the opportunity to work with them. The buffer which contains the stored integers, will work as a list in descending order and the implementation of the selection sort algorithm will then sort the list.
The list is split into a sorted and unsorted part where all elements are in the unsorted list at the beginning. The algorithm will then loop through the unsorted list and find the element with the smallest value. This element is then removed from the unsorted list and added to the end of the sorted list. When the whole list is sorted, each element is then printed into stdout.

2 Testing

The selection sort algorithm was tested with 50 different lists of numbers with the lengths 100, 1000, 5000, 10000 and 50000.
The entirety of the testing data can be found in the zip-file as Tests_on_data.txt.
After testing, it could be seen that the amount of comparisons made, was only determined by the length of the list as all similar tests yielded the same result.
The average runtime and million comparisons per second for each list is depicted in figure 1.

File size	100	1000	5000	10000	50000
Avg. time in s	0.004	0.0087	0.0312	0.082	0.7268
Avg. MCIPS	1.188	59.226	402.734	616.836	1720.529

Figure 1: Table of time & MCIPS

It can be concluded that the difference in MCIPS ratio from file size 100 to file size 1000 is the highest among the 5 averages. The larger the file size is the less significant this difference in ratio becomes.This development hinted towards a possible logistic development in the ratio between time and MCIPS. To test this a regression for a logistic development was made and depicted in figure 2.

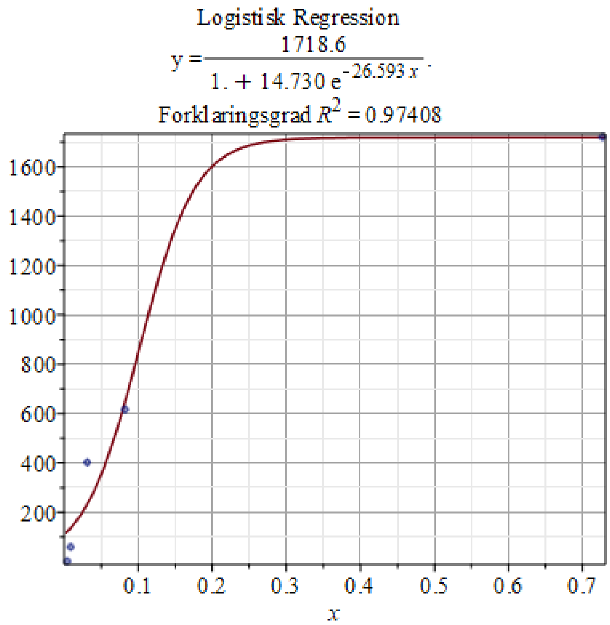


Figure 2: Logistic Regression

The closer R^2 is to 1.0 the better the model fits the data set. With a value for $R^2 = 0.97408$ implicates that the logistic regression is a good fit to the data. To further follow up on this we a single test with file size 1,000,000 and a single test with file size 2,000,000 was conducted. These results are depicted in figure 3.

File size	1,000,000	2,000,000
Time in s	246.5	1161
MCIPS	2028.4	1722.65

Figure 3: Table of time & MCIPS

Those results concluded that the MCIPS does not continually grow with bigger file sizes and converges to a value around 1720 MCIPS.

3 The results

From the test results it can be concluded that the input list(sorted or unsorted) doesn't affect the amount of comparisons made by the algorithm which is its runtime. The official runtime of selection sort is O(n^2) but the way selection is implemented in this project gave an amount of $< n^2$ comparisons made and it was consistent for each individual list length. When given a list with length (n), the algorithm looks through the unsorted list n times and makes p-1 comparisons(where p is the length of the unsorted list) with each loop. Because the unsorted list begins with the length n and decrements by 1 each loop, the total amount of comparisons made by the algorithm will be:

$$\sum_{k=1}^{n-1} k = \frac{n^2}{2} - \frac{n}{2}$$

In figure 3 the graph for this runtime is plotted with the graph for n^2 and it can be concluded that it's slightly faster. A decent runtime for a sorting algorithm is $n * log(n)$ which graph is also plotted, and shows that $\frac{n^2}{2} - \frac{n}{2}$ is still not an efficient time for a sorting algorithm.

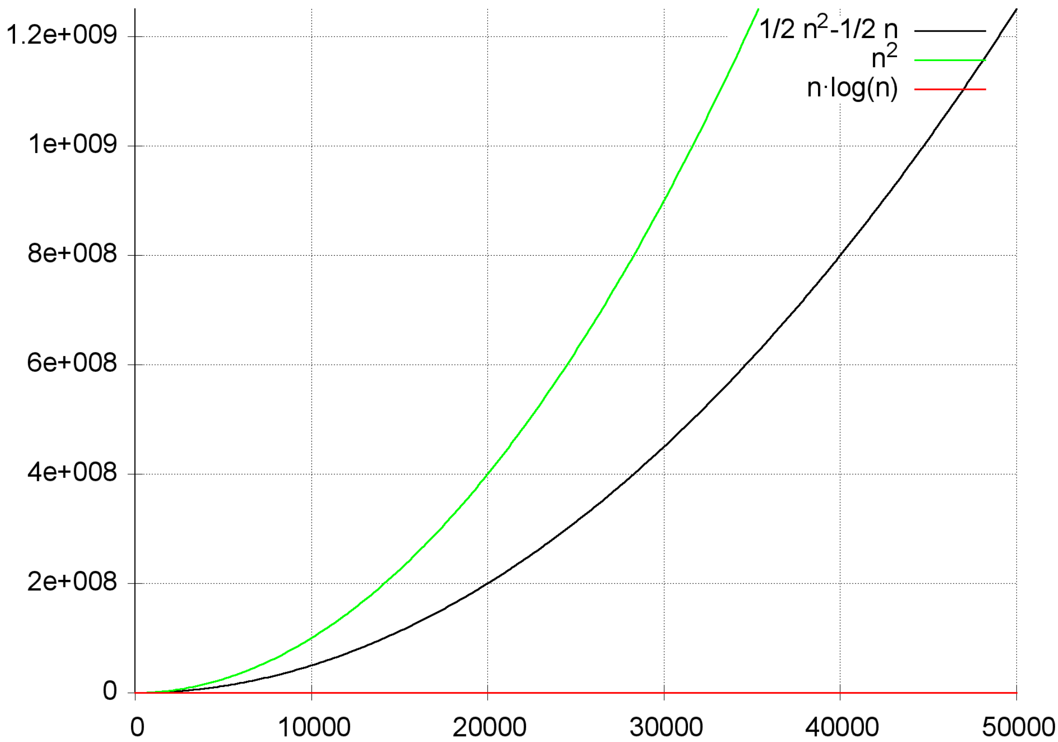


Figure 4: Grafen med n*n og n*log n

4 Source code

```
.section .data

.include "file_handling.asm"
.include "alloc.asm"
.include "parsing.asm"
.include "print.asm"

.section .text

.globl _start

_start:

mov 16(%rsp), %rdx     # Retrieve filename from command line argument

###########################################################
################### Open text file ########################
###########################################################
mov $2, %rax
mov %rdx, %rdi         # Pointer to a string (filename)
mov $0, %rsi           # Setting a flag, we use 0
mov $0, %rdx           # 0 is equal to read-only mode
syscall

push %rax              # Put File descriptor from rax to stack
call get_file_size     # Reads from the stack and returns filesize in rax
pop %r12               # Put File descriptor from the stack to r12

push %rax              # Put file size from rax to stack
call alloc_mem         # Reads from the stack and returns a pointer in rax
mov %rax, %r14         # Put the pointer to start of memory from rax to r14
pop %r13               # Put filesize in r13

###########################################################
#################### Read text file #######################
######## Copies data from the file to buffer (r14) ########
## Read the n bytes from the text file and return in rax ##
###########################################################
mov $0, %rax
mov %r12, %rdi         # file descriptor
mov %r14, %rsi         # pointer to memory
mov %r13, %rdx         # num of bytes to read (filesize)
syscall

push %r13              # push filesize to stack
push %r14              # push pointer to first memory buffer to stack
call get_number_count  # returns how many numbers is stored in buffer to rax
```

```
pop %r14                # pointer buffer 1
pop %r13                # filesize buffer 1

# amount of number times 8 gives us bytes needed for buffer 2
imul $8, %rax, %r9      # filesize buffer 2
push %r9                # Push filesize for buffer 2 to stack
call alloc_mem          # Returns a pointer to buffer 2 in rax

mov %rax, %r15          # Pointer to buffer 2 (start of memory)

push %r15               # Push start of buffer 2 on stack
push %r13               # Push filesize on stack
push %r14               # Push start of buffer 1 on stack
call parse_number_buffer # Writes ascii signs from buffer 1
                        # into integers in buffer 2
pop %r14                # pointer buffer 1
pop %r13                # filesize buffer 1
pop %r15                # pointer buffer 2
pop %r9                 # filesize buffer 2

############################################################
################## Selection Sort #########################
############## 1. Find minimum ############################
############## 2. Swap with first element #################
############## 3. Sort rest of the list ###################
############################################################
# rcx = pointer - above rcx list is sorted
# r8  = pointer to cmp value
# r9  = filesize buffer 2
# r10 = minimum value
# r12 = cmp value
# r13 = pointer to minimum
# r14 = counter
# r15 = pointer buffer 2
add %r15, %r9           # end of buffer
mov %r15, %rcx          # everything above rcx is sorted
# mov $0, %r14          # used as counter

outer:
  # outer for loop
  mov (%rcx), %r10      # First number is minimum
  mov %rcx, %r8         # r8 starts by pointing at first element
  inner:
    # inner for loop
    add $8, %r8         # r8 points to next number
    cmp %r8, %r9        # have we reached the end of the buffer?
    je endOfList        # exit inner for loop
    mov (%r8), %r12     # r12 is temporary compare value
    # inc %r14          # increments the counter
    cmp %r10, %r12      # is cmp value less than minimum?
    jl newMinimum       # if yes jump to newMinimum
    jge inner           # if no go to next number
              newMinimum:
      # overwrite r10 and r13 with the new minimum value and address
```

```
                mov %r12, %r10    # a new minimum value is saved
     mov %r8, %r13     # a new minimum address is saved
                jmp inner         # go to next number
endOfList:
  # The minimum of the unsorted list is found
       # We wish to put the minimum in top of the memory (first in the list)
  mov (%rcx), %r12      # moves first number in memory to minimum numbers address
  mov %r12, (%r13)
  mov %r10, (%rcx)      # moves minimum number to first numbers address
       add $8, %rcx          # we want minimum to be over rcx pointer
  cmp %rcx, %r9         # did rcx pointer reach end of buffer?
  jne outer             # if not, find another minimum

printing_loop:
# prints every number in buffer
push (%r15)
call print_number
pop %rax
add $8, %r15            # r15 points to next number
cmp %r15, %r9           # did r15 pointer reach end of buffer?
jne printing_loop       # if not, print another number

# push %r14
# call print_number     # prints the counter
# pop %r14

###########################################################
################### Close the file #######################
###########################################################
mov $3, %rax
mov $3, %rdi
syscall

###########################################################
################### syscall to exit #######################
###########################################################
mov $60, %rax
mov $0, %rdi
syscall
```