**Word Count Algorithms**

Marijn Verschuren  
510936

# **1. Table of contents**

[**1. Table of contents** 2](#_Toc1)

[**2. Code** 3](#_Toc2)

[2.1. Compare function 3](#_Toc3)

[2.2. Binary search 3](#_Toc4)

[2.3. Binary search 4](#_Toc5)

[2.4. Sorting 5](#_Toc6)

[3. Analysis 6](#_Toc7)

[3.1. Scalability (space complexity) 6](#_Toc8)

[3.2. Time complexity 6](#_Toc9)

[3.2.1. Linear 6](#_Toc10)

[3.2.2. Binary 6](#_Toc11)

[4. Unit tests 7](#_Toc12)

[5. Measurements 8](#_Toc13)

[6. Conclusion and notes 9](#_Toc14)

# **2. Code**

## 2.1. Compare function

; rax ->  
; rdi -> a  
; rsi -> b  
cmp\_words:  
 vmovdqu ymm0, [rdi] ; load 32 bytes of a  
 vmovdqu ymm1, [rsi] ; load 32 bytes of b  
 vpminub ymm2, ymm0, ymm1 ; \  
 vpcmpeqb ymm0, ymm0, ymm2 ; | compare all individual bytes of ymm0 and ymm1  
 vpcmpeqb ymm1, ymm1, ymm2 ; | result in ymm0 will contain 0xff if a < b, 0x00 if a == b and 0x01 if a > b  
 vpsubb ymm0, ymm0, ymm1 ; /  
 ; vectorized code...

This code compares two strings in a vectorized way it takes in two strings with a max length of 32. And returns -1 if word a is smaller than word b, 0 if they are equal and 1 if a is bigger than b. It runs 4 to 5 times faster than the C implementation.

## 2.2. Binary search

; rdi -> words -> cmp func a  
; rsi -> size -> cmp func b  
; rdx -> word  
; rcx -> cmp function  
; rax -> cmp return  
; rbx -> i  
; r8 -> words  
; r9 -> size  
; r10 -> flag  
; r11 -> count  
linear\_count:  
 xor rbx, rbx ; set i to 0  
 xor r11, r11 ; set count to 0  
 xor r10, r10 ; set the rest of the flag register to 0  
 mov r8, rdi ; move word array into r8  
 mov r9, rsi ; move size into r9  
 mov rsi, rdx ; move word into rsi  
 test r9, r9  
 jz lc\_end  
 lc\_loop:  
 mov rdi, [r8 + 8 \* rbx] ; load words[i] into rdi  
 call rcx ; call compare words on rdi, rsi  
 test rax, rax ; test rax  
 setz r10b ; conditionally set r10 to 1  
 add r11, r10 ; add r10 to the count  
 inc rbx ; increment i  
 cmp rbx, r9 ; test i < size  
 jl lc\_loop  
 lc\_end:  
 mov rax, r11 ; move count into rax  
 ret

This code takes in an array of words, a size of this array, a word to count and a compare function and will return the amount of occurrences of the word in the array.

## 2.3. Binary search

; rdi -> words -> cmp func a  
; rsi -> size -> cmp func b  
; rdx -> word  
; rcx -> cmp function  
; rax -> cmp return  
; rbx -> i  
; r8 -> words  
; r9 -> size  
; r10 -> lower\_bound  
; r11 -> upper\_bound  
; r12 -> upper\_bound copy  
binary\_count:  
 xor rax, rax ; clear return value  
 mov r8, rdi ; word array  
 mov r9, rsi ; word array size  
 xor r10, r10 ; clear r10  
 mov r11, r9 ; mov size into r11  
 dec r11 ; calculate size - 1  
 mov rsi, rdx ; word a  
 test r9, r9 ; check if there are elements in the word array  
 jz bc\_end ; return if not  
 bc\_loop:  
 mov rbx, r10 ; move the lower\_bound into rbx  
 add rbx, r11 ; add the upperbound to rbx  
 shr rbx, 1 ; divide rbx by 2  
 mov rdi, [r8 + 8 \* rbx] ; load words[i] into rdi  
 call rcx ; call compare words on rdi, rsi  
 test rax, rax ; test rax  
 jz bc\_hit ; jump to hit subroutine if words match  
 cmovp r10, rbx ; move on parity even (-1 -> 0b11111111)  
 cmovnp r11, rbx ; move on parity odd (-1 -> 0b11111111)  
 ; *TODO: improve compare* mov r12, r11 ; copy upper\_bound to r12  
 sub r12, r10 ; subtract lower\_bound  
 cmp r12, 1 ; check if the difference is greater than 1  
 jg bc\_loop  
 xor rax, rax ; clear return value if the word was not found  
 ret  
 bc\_hit:  
 mov r10, rbx ; move the hit location into lower\_bound  
 bc\_hit\_loop\_low:  
 dec r10 ; decrement lower\_bound  
 test r10, r10 ; check if lower\_bound is still greater than 0  
 js bc\_hit\_loop\_low\_end ; jump to loop end if not  
 mov rdi, [r8 + 8 \* r10] ; load words[i] into rdi  
 call rcx ; call compare words on rdi, rsi  
 test rax, rax ; test rax  
 jz bc\_hit\_loop\_low  
 bc\_hit\_loop\_low\_end:  
 mov r11, rbx ; move the hit location into upper\_bound  
 bc\_hit\_loop\_high:  
 inc r11 ; increment upper\_bound  
 cmp r11, r9 ; compare upper\_bound to array size  
 je bc\_hit\_loop\_high\_end ; jump to loop end if they are equal  
 mov rdi, [r8 + 8 \* r11] ; load words[i] into rdi  
 call rcx ; call compare words on rdi, rsi  
 test rax, rax ; test rax  
 jz bc\_hit\_loop\_high  
 bc\_hit\_loop\_high\_end:  
 mov rax, r11 ; move upper\_bound into return value  
 sub rax, r10 ; subtract lower\_bound from return value  
 dec rax ; decrement return value to correct for offset  
 bc\_end:  
 ret

This code takes and returns the same values as the linear variant (given that the word array is sorted) but finds the words in a different way. It uses the widely used binary search algorithm to find the words and runs almost twice as fast as the linear variant on a 239 KB file.

## 2.4. Sorting

; rax -> ret from cmp  
; rbx -> j  
; rcx -> i  
; rdx -> cmp function -> unused  
; rsi -> arg 1 for cmp (array[i])  
; rdi -> arg 0 for cmp (array[i-1])  
; r8 -> array  
; r9 -> length  
; r10 -> length - j  
; r11 -> cmp function (rbx is cleared after calling cmp function)  
sort:  
 mov r8, rax ; move array to r8  
 mov r9, rsi ; move length into r9  
 mov r11, rdx ; move cmp function into r11  
 xor rbx, rbx ; uint64\_t j = 0;  
 loop\_0:  
 mov rcx, 1 ; uint64\_t i = 1;  
 mov r10, r9 ; move the array length into rdi  
 sub r10, rbx ; calculate (length - j)  
 loop\_1:  
 mov rdi, [r8 + rcx \* 8]  
 mov rsi, [r8 + (rcx - 1) \* 8]  
 call r11  
 test rax, rax  
 cmovp rdi, rsi ; move on parity even (-1 -> 0b11111111)  
 cmovp rsi, [r8 + rcx \* 8] ; move on parity even (-1 -> 0b11111111)  
 mov [r8 + (rcx - 1) \* 8], rsi  
 mov [r8 + rcx \* 8], rdi  
 inc rcx ; i++  
 cmp rcx, r10 ; compare (i) and (length - j)  
 jl loop\_1 ; jump only if (i) is less than (length - j)  
 inc rbx ; j++  
 cmp rbx, r9 ; compare (j) and (length)  
 jl loop\_0 ; jump only of (j) is less than (length)  
 ret

This code implements bubble sort using a compare function

# 3. Analysis

## 3.1. Scalability (space complexity)

The scalability is entirely dependent on the space complexity of the algorithm which is O(n) in both cases because they don’t make any copies of the array.

## 3.2. Time complexity

### 3.2.1. Linear

Before we can analyze the linear search algorithm we need to analyze the function that splits the text into an array of words. The function will read the whole text once to determine the word count this will start us of at O(n) then we go back to the start of the array where we start reading the words. Since there are no other operations preformed other than a copy the complexity will be: O(2n + 5rm) where ‘n’ is the total amount of characters in the text, ‘r’ is the complexity of the copy function and ‘m’ is the amount of words (5 is an estimate of the average length of an English word). The expression can be simplified to O(2rn) because ‘m’ will be approximately ‘n’ / 5. Lets assume the complexity of the copy function is O(1) leaving us with a complexity of O(2n) => O(n).

For the linear search function the complexity is O(rn) where ‘r’ is the complexity of the compare function and ‘n’ is the amount of words in the array. If we look at the compare function at [2.1. Compare function](#_2.1._Compa) we see that it excepts words shorter than 32 in exchange for a complexity of O(1) thanks to SIMD instructions. This leaves us with a complexity of O(n)

Over all the complexity becomes O(2n + n) => O(n)

### 3.2.2. Binary

Like with the [linear](#_3.2.1._Lin) variant we need to split the text into an array of words which has a complexity of O(2n) => O(n).

For the binary search function the word array has to be sorted this is done using the sorting function found at [2.4. Sorting](#_2.4._Sorti) implementing the bubble sort algorithm, with the complexity of O(rn(n-1)) where ‘r’ is the complexity of the compare function which we established to be O(1) and ‘n’ the number of elements. When simplifying the expression we get O(n^2 – n) => O(n^2)

For the binary search function it self the complexity is O(r log2(n)) where ‘r’ is once again the complexity of the compare function: O(1) and ‘n’ the number of words in the array. The log2 originates from the fact that we can rule out half of the array at every compare step, greatly increasing efficiency. For the function itself the complexity becomes O(log2(n)).

When using the binary search algorithm once will result in the following complexity:   
O(2n + n^2 – n + log2(n)) which simplifies to O(n^2 + n + log2(n)) => O(n^2) this is due to the fact that the array has to be sorted.

Luckily this has to be done only once leaving us with a complexity of O(log2(n)) for all following calls.

# 4. Unit tests

## 4.1. Compare functions

str padded\_word\_a = calloc(sizeof(char), PADDING);  
str padded\_word\_b = calloc(sizeof(char), PADDING);  
memcpy(padded\_word\_a, "1234567890", 11);  
memcpy(padded\_word\_b, "1234567891", 11);  
printf("compare\_words\t(null, null) == 0\t->\t%d\n", compare\_words(nullptr, nullptr) == 0);  
printf("cmp\_words\t\t(null, null) == 0\t->\t%d\n", cmp\_words(nullptr, nullptr) == 0);  
printf("compare\_words\t(eq, eq) == 0\t\t->\t%d\n", compare\_words("1234567890", "1234567890") == 0);  
printf("cmp\_words\t\t(eq, eq) == 0\t\t->\t%d\n", cmp\_words(padded\_word\_a, padded\_word\_a) == 0);  
printf("compare\_words\t(lt, gt) == -1\t\t->\t%d\n", compare\_words("1234567890", "1234567891") == -1);  
printf("cmp\_words\t\t(lt, gt) == -1\t\t->\t%d\n", cmp\_words(padded\_word\_a, padded\_word\_b) == -1);  
printf("compare\_words\t(gt, lt) == 1\t\t->\t%d\n", compare\_words("1234567891", "1234567890") == 1);  
printf("cmp\_words\t\t(gt, lt) == 1\t\t->\t%d\n", cmp\_words(padded\_word\_b, padded\_word\_a) == 1);

Here the correctness of the compare functions is tested (compare words is a C implementation to test against). Keep in mind that the cmp\_words have to be padded to facilitate the vectorized check.

Result:

compare\_words (null, null) == 0 -> 1  
cmp\_words (null, null) == 0 -> 1  
compare\_words (eq, eq) == 0 -> 1  
cmp\_words (eq, eq) == 0 -> 1  
compare\_words (lt, gt) == -1 -> 1  
cmp\_words (lt, gt) == -1 -> 1  
compare\_words (gt, lt) == 1 -> 1  
cmp\_words (gt, lt) == 1 -> 1

## 4.2. Search functions

// normal text case  
str\* words;  
uint64\_t size = text\_to\_words("I'm trying to check if a a a value is zero in x86\_64 assembly code zero zero zero", &words, PADDING);  
str padded\_word = malloc(PADDING);  
memset(padded\_word, 0x00, PADDING);  
memcpy(padded\_word, "zero", 5);  
sort(words, size, (comp\_fn)cmp\_words);  
printf("linear\_count (words, size, padded\_word, cmp\_words) == 4\t->\t%d\n", linear\_count((const str\*)words, size, padded\_word, (comp\_fn)cmp\_words) == 4);  
printf("binary\_count (words, size, padded\_word, cmp\_words) == 4\t->\t%d\n", binary\_count((const str\*)words, size, padded\_word, (comp\_fn)cmp\_words) == 4);  
for (uint64\_t i = 0; i < size; i++) {  
 free(words[i]);  
} free(words);  
// empty case  
printf("linear\_count (null, 0, null, null) == 0\t->\t\t\t\t\t%d\n", linear\_count(nullptr, 0, nullptr, (comp\_fn)nullptr) == 0);  
printf("binary\_count (null, 0, null, null) == 0\t->\t\t\t\t\t%d\n", binary\_count(nullptr, 0, nullptr, (comp\_fn)nullptr) == 0);

Here the correctness of the search functions is tested.

Result:

linear\_count (words, size, padded\_word, cmp\_words) == 4 -> 1  
binary\_count (words, size, padded\_word, cmp\_words) == 4 -> 1  
linear\_count (null, 0, null, null) == 0 -> 1  
binary\_count (null, 0, null, null) == 0 -> 1

# 5. Measurements

## 5.1. Compare functions

clock\_t start, end;  
start = clock(); // start timer  
for (uint64\_t i = 0; i < 10000000; i++) {  
 compare\_words("1234567890", "1234567891");  
}  
end = clock(); // stop timer  
printf("time:\t\t%f\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);  
start = clock(); // start timer  
for (uint64\_t i = 0; i < 10000000; i++) {  
 cmp\_words(padded\_word\_a, padded\_word\_b);  
}  
end = clock(); // stop timer  
printf("time:\t\t%f\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

This code records the time it takes both functions to compare 10 million words.

Result:

compare\_words: 0.374989  
cmp\_words: 0.031454

In this result the benefits of SIMD can be seen VERY clearly.

## 5.2. Search and sort functions

start = clock(); // start timer  
sort(words, size, (comp\_fn)cmp\_words);  
end = clock(); // stop timer  
printf("sort:\t\t\t\t%f\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);  
start = clock(); // start timer  
for (uint64\_t i = 0; i < 100; i++) {  
 linear\_count((const str\*) words, size, padded\_word, (comp\_fn) cmp\_words);  
}  
end = clock(); // stop timer  
printf("linear search:\t\t%f\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);  
start = clock(); // start timer  
for (uint64\_t i = 0; i < 100; i++) {  
 binary\_count((const str\*)words, size, padded\_word, (comp\_fn)cmp\_words);  
}  
end = clock(); // stop timer  
printf("binary search:\t\t%f\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

This code records the time it takes the sort function to sort a list with words from a 239 KB file. Then the time to search 100 words in this sorted array is recorded.

Result:

sort: 0.882636  
linear search: 0.006745  
binary search: 0.003655

Here you can clearly see the impact a complexity O(n^2) (sorting) has on execution time.

# 6. Conclusion and notes

In conclusion, use binary search when the sorted array can be reused to yield the very efficient complexity of O(log2(n)). Otherwise the linear search should be used to avoid the massive execution time of sorting the array with complexity O(n^2) (this can be reduced by using a better algorithm but will never be less than O(n log(n))). Furthermore the use of SIMD can in some cases improve the complexity to compare from O(n) to O(1) given that no words will exceed a length of 32, greatly improving performance once more.

One problem I found during this assignment is that I implemented binary\_count in a slightly different way. This is because when I was working on a tree implementation one of my classmates pointed to the uselessness of using trees to which my response was to remove trees altogether having forgot the details of the assignment after reading it a couple of days before.