# Harry Stern

## CS 211 Assignment 1

My algorithm works as follows. It runs in $O(n \log n)$ time and uses $O(n)$ space.

First, we read the file into a string. As we traverse the string, we keep track of whether we are inside or outside a word, and malloc a new word every time we leave from the inside to the outside. This new word gets appended to what is essentially an arraylist (`wordlist *wl`). This part takes $O(n)$ time, where $n$ is the number of characters in the file, because the amortized cost of appending to the array takes O(1) since we keep track of the end.

At most we can create $\frac{n}{2}$ strings from the previous operation, if, for example, we alternated 'a' and space characters the whole file. Therefore, for the rest of the analysis $n$ will be the number of strings, but since it is $O(n)$ of the number of characters, the analysis stays consistent.

Next, we sort the array using `qsort`, the builtin quicksort implementation from `stdlib.h`. Using a custom comparison function (`int compare_strings_lower(const void *a, const void *b)`), we malloc two strings and copy a and b into them, making each character lowercase using `tolower` from `ctypes.h`. Then we immediately free the new strings after comparison. I believe, but I am not sure, that this adds at most only an extra $O(n)$ number of operations to quicksort's $O(n \log n)$, making the entire sorting step still $O(n \log n)$.

This step could be optimized the most. The builtin quicksort may or may not be optimized to always run in $O(n \log n)$, so we could write a mergesort or some other sort that is always that fast. Furthermore, the rapid allocation and freeing of the lowercase strings could be replaced by writing our own version of `strcmp` from `string.h`. The memory usage isn't that important because it it immediately freed, so only two extra strings are ever allocated at a time for the duration of the sort, but the calls themselves and the copies take time.

We then traverse this list inside our `occurlist *ol_create(wordlist *wl)` function, creating a `wordoccur *wl` whenever we reach a word that hasn't occurred before. We know that we only need to traverse the list once, because the words are sorted lexicographically without taking into account case. If they were not sorted without taking the lowercase version, situations like "aab", "ABA", "AAB" would make it so that we'd have to search through the occurrence list each time to make sure the lowercase version hasn't occurred previously. We do have to compare each of the previous versions of the current occurrence so we can count the number of unique versions properly. I believe this step also adds at most $O(n)$ to the total number of operations. Therefore this step is also $O(n)$.

Finally, we print the results by traversing the array of structs containing the occurrences. $O(n)$. Since the longest step was $O(n \log n)$ and there were only a constant number of $O(n)$ steps, the whole thing runs in $O(n \log n)$, which seems to make sense given that I can run it on the ilabs' /usr/share/dict/words, which has 479829 lines, in 1.340 seconds.

Regarding memory usage, the string holding the contents of the file is $O(n)$ by definition, the list holding the individual words is $O(n)$, and if each individual string is unique, we also need $O(n)$ to store them in the occurrences list. We also need some temporary memory to hold lowercase versions of the strings when we compare in the sorting step, which may be as large as $\frac{n}{2}$ if the file consists of only two long strings. Therefore, we use $O(n)$ space at runtime. Furthermore, I used valgrind to locate and debug all memory leaks, double frees, and out of bounds writes.

Here are the values of `info registers` at the beginning of the functions `void parse_words(char *text, wordlist *wl);` and `void ol_print(occurlist *ol);`, which are the equivalents of `processStr` and `printResult` in my program.

```
Breakpoint 1, parse_words (
15 int state = OUT;
(gdb) info registers
rax            0x7ffff758f010 140737343189008
rbx            0x0 0
rcx            0x0 0
rdx            0x603010 6303760
rsi            0x603010 6303760
rdi            0x7ffff758f010 140737343189008
rbp            0x7fffffffe370 0x7fffffffe370
rsp            0x7fffffffe330 0x7fffffffe330
r8             0x7ffff7fbe700 140737353869056
r9             0x7ffff7fbe700 140737353869056
r10            0x7fffffffe0c0 140737488347328
r11            0x206 518
r12            0x4009a0 4196768
r13            0x7fffffffe490 140737488348304
r14            0x0 0
r15            0x0 0
rip            0x400a94 0x400a94 <parse_words+16>
eflags         0x206 [ PF IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0 0
es             0x0 0
fs             0x0 0
gs             0x0 0
(gdb) continue
```

```
Continuing.

Breakpoint 2, ol_print (ol=0x16110c0) at occurlist.c:70
70 printf("Word Total No. Occurences No. Case-Sensitive Versions\n");
(gdb) info registers
rax            0x16110c0 23138496
rbx            0x0 0
rcx            0x6740ea0 108269216
rdx            0x66393 418707
rsi            0x4 4
rdi            0x16110c0 23138496
rbp            0x7fffffffe370 0x7fffffffe370
rsp            0x7fffffffe340 0x7fffffffe340
r8             0x7ffff7dd8ee8 140737351880424
r9             0x1 1
r10            0x4 4
r11            0x206 518
r12            0x4009a0 4196768
r13            0x7fffffffe490 140737488348304
r14            0x0 0
r15            0x0 0
rip            0x401272 0x401272 <ol_print+13>
eflags         0x202 [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0 0
es             0x0 0
fs             0x0 0
gs             0x0 0
```