In my zip extension, I tried to optimize my implementation of deflate as much as I could for speed.

Here are some Initial Benchmarks of deflate:
I compressed and decompressed the text moby dick

My implementation

	Moby dick 2 times	4 times	8 times
Trial 1	7.76	24.946s	1m 54s
Trial 2	7.436	26.522s	1m 50s
Trial 3	7.359	29.323s	1m 52s

Their implementation

	Moby dick 2 times	4 times	8 times
Trial 1	.249s	.285s	.6s
Trial 2	.142s	.258s	.52s
Trial 3	.145s	.331s	.54s

Note that my implementation is somehow quadratic in time.

A lot of the time I spent was setting up the whole profiler, because I couldn't figure out how to use tools like perf on my WSL environment. For all of these, I just used callgrind. Profiling even 1 copy of moby dick takes too long for my implementation to profile, so I shorten it to only 5000 lines. Here is the output of the profiler before any optimizations.

100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.98	0.00	1	(below main)	simple_compress
99.98	0.00	1	__libc_start_main@@GLIBC_2.34	libc.so.6: libc-start.c
99.98	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.98	0.00	1	main	simple_compress
99.98	0.00	1	std::rt::lang_start	simple_compress: rt.rs
99.98	0.00	1	std::rt::lang_start_internal	simple_compress: rt.rs, mod.rs, intrinsics.r...
99.96	0.00	1	std::rt::lang_start::{{closure}}	simple_compress: rt.rs
99.96	0.00	1	std::sys_common::backtrace::__rust_begin_short_backtrace	simple_compress: backtrace.rs, hint.rs
99.96	0.00	1	core::ops::function::FnOnce::call_once	simple_compress: function.rs
99.96	0.00	1	simple_compress::main	simple_compress: simple_compress.rs
99.96	0.00	1	utils::huffman::compress	simple_compress: huffman.rs
99.96	0.12	1	utils::huffman::lz77_compression	simple_compress: huffman.rs
91.97	5.95	5 858	utils::huffman::find_match_buffer	simple_compress: huffman.rs
54.99	1.46	268 364	std::collections::hash::map::HashMap<K,V,S>::entry	simple_compress: map.rs
53.53	2.29	268 364	hashbrown::rustc_entry::<impl hashbrown::map::HashMap<...	simple_compress: rustc_entry.rs, map.rs
36.81	0.76	304 809	core::hash::BuildHasher::hash_one	simple_compress: mod.rs
18.58	0.25	304 809	core::hash::impls::<impl core::hash::Hash for &T>::hash	simple_compress: mod.rs
18.33	0.83	304 809	core::hash::impls::<impl core::hash::Hash for (T,B,C)>::hash	simple_compress: mod.rs
18.26	4.53	280 077	hashbrown::raw::RawTable<T,A>::find	simple_compress: mod.rs, sse2.rs, non_nul...
17.50	1.08	914 427	core::hash::impls::<impl core::hash::Hash for u8>::hash	simple_compress: mod.rs
16.42	1.08	914 427	core::hash::Hasher::write_u8	simple_compress: mod.rs
15.34	0.86	914 427	<std::hash::random::DefaultHasher as core::hash::Hasher>::...	simple_compress: random.rs, sip.rs

Most stuff is in find match buffer, spending half the time in with hashmap optimizations

After going through the code, I was doing something silly like checking if the value was not in an array, then appending it to the top of list. This can be done in O(1) with a hashSet, instead of 2 O(n) operations. This would happen every time we call

After hashset

2 moby dicks	4 moby dicks	8 moby dicks
7.507 s	25.5 s	1m 28 sec

There is a slight performance increase, but somehow the timing is still quadratic. I'm assuming running into the Hashsets worst case performance a lot of the time. This is really surprising to me because I was always told that hashsets are O(1), but clearly this is not the case all the time. However, this was just a problem with our algorithm.

In the algorithm, we have to iterate through all the indices. This led me to clear the hashset as we will have lots of unused indices. We know that if the index that the match is at is already too large, we can simply delete the index from the list entirely, and therefore, we don't have to iterate over them. This led to a huge performance increase, and actually made my implementation run in linear time with respect to the number of moby dicks I compressed. It still isn't looking anywhere close to miniz_oxides implementation of zip though.

2 moby dicks	4 moby dicks	8 moby dicks
3.57s	7.406s	14.7s

Even after that optimization, the profiler results look the same, we're spending a large majority of our time in hashing functions, and doing stuff with hashmaps

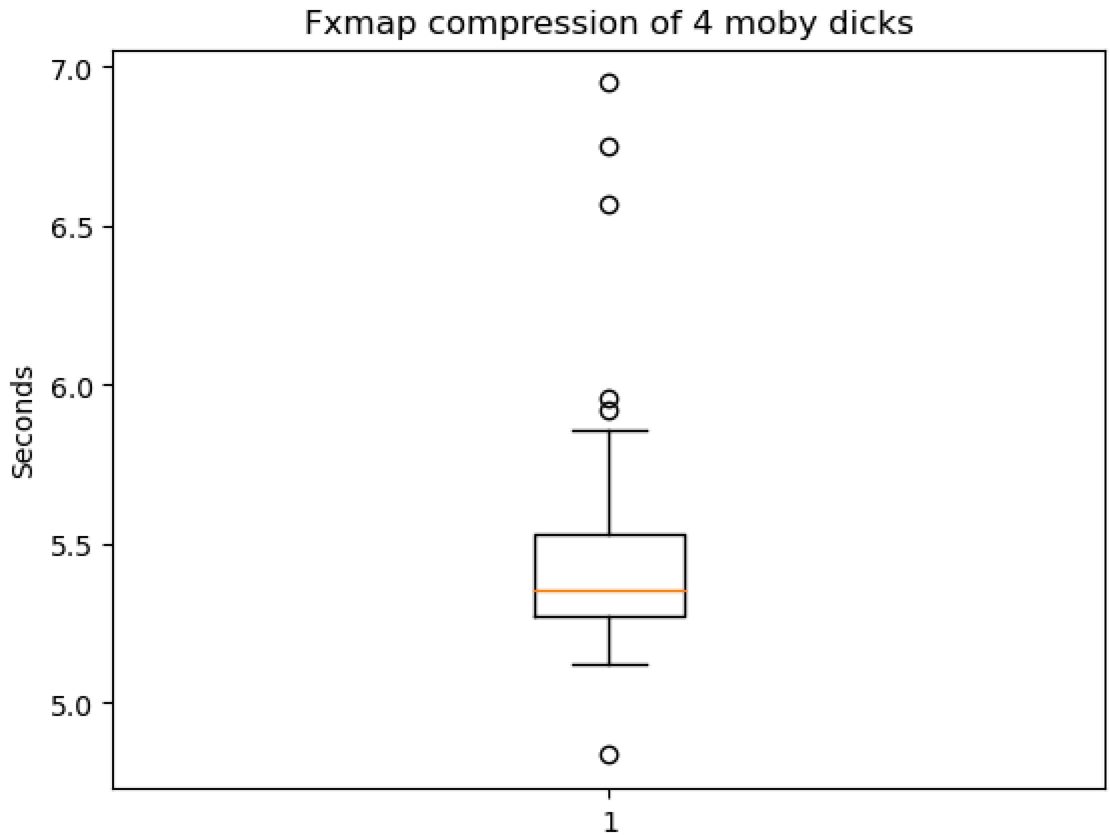
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
100.00	0.00	1	(below main)	simple_compress
100.00	0.00	1	__libc_start_main@@GLIBC_2.34	libc.so.6: libc-start.c
100.00	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
100.00	0.00	1	main	simple_compress
100.00	0.00	1	std::rt::lang_start	simple_compress: rt.rs
100.00	0.00	1	std::rt::lang_start_internal	simple_compress: rt.rs, mod.rs, intrinsics.r…
100.00	0.00	1	std::rt::lang_start::{{closure}}	simple_compress: rt.rs, process.rs, proces…
100.00	0.00	1	std::sys_common::backtrace::__rust_begin_short_backtrace	simple_compress: backtrace.rs, hint.rs
100.00	0.00	1	core::ops::function::FnOnce::call_once	simple_compress: function.rs
100.00	0.00	1	simple_compress::main	simple_compress: simple_compress.rs, rt.rs
99.81	0.16	1	utils::huffman::compress	simple_compress: huffman.rs
99.06	0.03	1	utils::huffman::lz77_compression	simple_compress: huffman.rs
96.43	4.18	31 958	utils::huffman::find_match_buffer	simple_compress: huffman.rs, option.rs
43.33	1.04	8 624 634	core::hash::BuildHasher::hash_one	simple_compress: mod.rs
39.56	1.08	4 081 429	std::collections::hash::map::HashMap<K,V,S>::entry	simple_compress: map.rs
38.49	1.68	4 081 429	hashbrown::rustc_entry::<impl hashbrown::map::HashMap<K,V,…	simple_compress: rustc_entry.rs, map.rs
34.38	1.60	4 177 297	hashbrown::map::HashMap<K,V,S,A>::insert	simple_compress: map.rs, mut_ptr.rs, mod…
33.51	0.44	4 081 429	std::collections::hash::set::HashSet<T,S>::insert	simple_compress: set.rs, set.rs, option.rs
18.34	0.35	8 624 634	core::hash::impls::<impl core::hash::Hash for &T>::hash	simple_compress: mod.rs
17.78	0.30	8 624 634	<std::hash::random::DefaultHasher as core::hash::Hasher>::fini…	simple_compress: random.rs, sip.rs
17.49	1.68	8 624 634	<core::hash::sip::Hasher<S> as core::hash::Hasher>::finish	simple_compress: sip.rs
15.46	0.79	17 320 412	<std::hash::random::DefaultHasher as core::hash::Hasher>::write	simple_compress: random.rs, sip.rs
14.98	1.37	4 177 297	hashbrown::raw::RawTable<T,A>::find_or_find_insert_slot	simple_compress: mod.rs, non_null.rs, mut…
14.66	5.32	17 320 412	<core::hash::sip::Hasher<S> as core::hash::Hasher>::write	simple_compress: sip.rs, intrinsics.rs, cmp.…
13.55	3.37	4 330 036	hashbrown::raw::RawTable<T,A>::find	simple_compress: mod.rs, sse2.rs, non_nul…
12.68	0.57	4 347 889	core::hash::impls::<impl core::hash::Hash for (T,B,C)>::hash	simple_compress: mod.rs
12.10	0.75	13 043 667	core::hash::impls::<impl core::hash::Hash for u8>::hash	simple_compress: mod.rs
11.76	11.76	8 624 634	<core::hash::sip::Sip13Rounds as core::hash::sip::Sip>::d_rounds	simple_compress: sip.rs, mod.rs
11.36	0.75	13 043 667	core::hash::Hasher::write_u8	simple_compress: mod.rs
11.32	2.34	4 177 297	hashbrown::raw::RawTableInner::find_or_find_insert_slot_inner	simple_compress: mod.rs, sse2.rs, non_nul…

This is surprising to me because I thought that the zip implementation would be a lot faster than mine because it does weird bitwise stuff that I didn't think of, but I guess the main reason is that their hashmap is just smarter.

I tried swapping out my hashmap with fxhash map, because I just looked it up and it's a faster hashmap. Here are the compression speeds with 4 moby dicks. I also ran the script 55 times, to get a better idea of the spread.

Compressing Moby dick 4 times

5.241s	5.245s	5.313s



Average time: 5.4731454545454525

I realized I can switch out the Hashset with the fxhashset as well, and found an even bigger performance increase, averaging around 2.4 seconds for compressing moby dick 4 times

Compressing Moby dick 4 times

2.680s	2.442s	2.574s

2.6356200000000003

Profiling the hashset and map, we get

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
100.00	0.00	1	(below main)	simple_compress
100.00	0.00	1	__libc_start_main@@GLIBC_2.34	libc.so.6: libc-start.c
100.00	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
100.00	0.00	1	main	simple_compress
100.00	0.00	1	std::rt::lang_start	simple_compress: rt.rs
100.00	0.00	1	std::rt::lang_start_internal	simple_compress: rt.rs, mod.rs, intrinsics.rs, stack_overflow.rs, ...
100.00	0.00	1	std::rt::lang_start::{{closure}}	simple_compress: rt.rs, process.rs, process_common.rs
100.00	0.00	1	std::sys_common::backtrace::__rust_begin_short_backtr...	simple_compress: backtrace.rs, hint.rs
100.00	0.00	1	core::ops::function::FnOnce::call_once	simple_compress: function.rs
100.00	0.00	1	simple_compress::main	simple_compress: simple_compress.rs, rt.rs
99.67	0.27	1	utils::huffman::compress	simple_compress: huffman.rs
98.40	0.05	1	utils::huffman::lz77_compression	simple_compress: huffman.rs
94.51	7.09	31 958	utils::huffman::find_match_buffer	simple_compress: huffman.rs, option.rs
34.09	1.82	4 081 429	std::collections::hash::map::HashMap<K,V,S>::entry	simple_compress: map.rs
32.26	2.85	4 081 429	hashbrown::rustc_entry::<impl hashbrown::map::Hash...	simple_compress: rustc_entry.rs, map.rs
31.13	2.71	4 177 297	hashbrown::map::HashMap<K,V,S,A>::insert	simple_compress: map.rs, mut_ptr.rs, mod.rs, non_null.rs, mod...
30.43	0.75	4 081 429	std::collections::hash::set::HashSet<T,S>::insert	simple_compress: set.rs, set.rs, option.rs
24.73	2.33	4 177 297	hashbrown::raw::RawTable<T,A>::find_or_find_insert_slot	simple_compress: mod.rs, non_null.rs, mut_ptr.rs
22.88	5.69	4 330 036	hashbrown::raw::RawTable<T,A>::find	simple_compress: mod.rs, sse2.rs, non_null.rs, mut_ptr.rs, bitm...
19.17	3.97	4 177 297	hashbrown::raw::RawTableInner::find_or_find_insert_slot...	simple_compress: mod.rs, sse2.rs, non_null.rs, mut_ptr.rs, bitm...
11.60	1.84	8 624 853	core::hash::BuildHasher::hash_one	simple_compress: mod.rs
9.80	6.02	22 950 408	<alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::...	simple_compress: mod.rs, raw_vec.rs, const_ptr.rs, metadata.r...
8.83	0.59	8 624 853	core::hash::impls::<impl core::hash::Hash for &T>::hash	simple_compress: mod.rs

And the profiler still says the main reason it's slow is because of the hashmaps and sets. If I want to get any faster, I would need to reduce lookups in hashmaps, or implement a more effective hashmap. I didn't know the most effective ways, so I looked at the source code of miniz_oxide.

To understand how they got away with this, I realized that miniz_oxide uses their own custom hashmap, that lets them do rolling hash functions so that the hashes aren't recomputed everytime, and that they can look things up really easily. In my case, I'm doing an insert into a hashset. Since the hashset isn't ordered I have to go through every value, and check if it's the less distance and a longer match, but if I use the rolling hash stuff, I know the order,

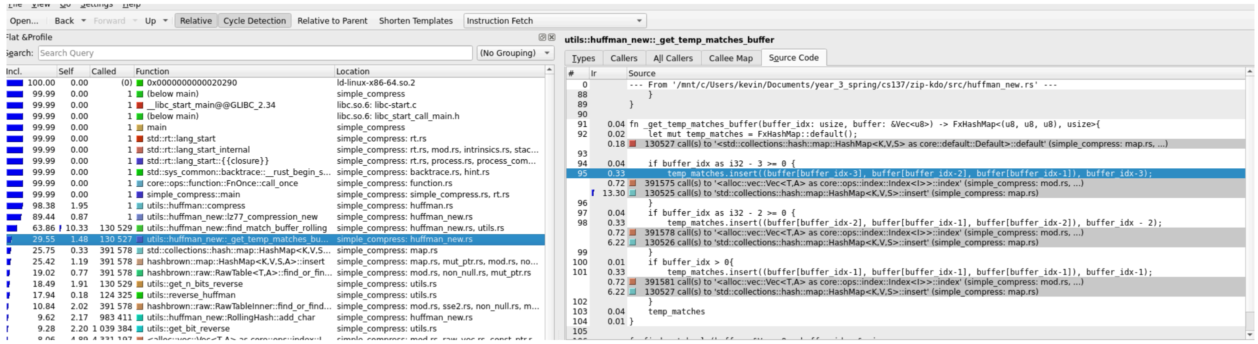
After lots of investigating, I realized why I was using a hashset in the first place, while the miniz_oxide implementation can get away with not using a hashset. I was first thinking that since we're always doing that appending to the front of the list, we can use a linked list

Doing this improvement even shortened the compression length a bit, around 10%. I assume it's because we're now actually taking the closest match, which is compressed with less bits. I'm honestly not sure how I passed some of the zip tests if that was the case.

With this, I was finally able to break the 1 second barrier for the compression of 2 moby dicks. Now the profiler is telling me a huge bottle neck is converting the linked list into an iterable and when I do the clone.

Copying inspiration from miniz_oxide, I used a custom hashmap. They also use a rolling hashmap, to compute the values faster. Although we only compare 3 values, instead of computing 3 different hashes, we just add the last element, and subtract the first element. We also know that we only have to store the max distance amount of numbers at any time.
https://en.wikipedia.org/wiki/Rabin–Karp_algorithm

This significantly improved our performance. In fact, now functions like Get temp matches buffer takes a significant amount of time, which was just making a hashmap to do run length encoding.



And I cleaned that up, by completely disregarding the FxHashMap entirely. Instead I just use an array. At this point speed ups are not really noticeable.

I then implemented multithreading, which splits the blocks into 32k blocks that can be decoded separately. This took a bit to learn. This takes a minor hit in the compression performance, but gets my implementation almost to the same level as gzip.

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.99	0.00	1	(below main)	simple_compress
99.99	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.99	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.99	0.00	1	main	simple_compress
99.99	0.00	1	std::rt::lang_start	simple_compress: rt.rs
99.99	0.00	1	std::rt::lang_start_internal	simple_compress: rt.rs, mod.rs, intrinsics.rs, stack_overflo...
99.98	0.00	1	std::rt::lang_start::{{closur...	simple_compress: rt.rs, process.rs, process_common.rs
99.98	0.00	1	std::sys_common::backtrac...	simple_compress: backtrace.rs, hint.rs
99.98	0.00	1	core::ops::function::FnOnce...	simple_compress: function.rs
99.98	0.00	1	simple_compress::main	simple_compress: simple_compress.rs, rt.rs
97.65	2.85	1	utils::huffman::compress	simple_compress: huffman.rs
84.59	1.28	1	utils::huffman_new::lz77_c...	simple_compress: huffman_new.rs
50.06	13.61	131 560	utils::huffman_new::find_m...	simple_compress: huffman_new.rs, utils.rs
24.25	2.81	131 560	utils::get_n_bits_reverse	simple_compress: utils.rs
23.62	0.26	125 598	utils::reverse_huffman	simple_compress: utils.rs
15.66	3.42	1 063 125	utils::huffman_new::Rolling...	simple_compress: huffman_new.rs
13.61	3.23	1 047 869	utils::get_bit_reverse	simple_compress: utils.rs
12.24	2.42	668 448	utils::huffman_new::Rolling...	simple_compress: huffman_new.rs
11.99	7.27	4 432 098	<alloc::vec::Vec<T,A> as c...	simple_compress: mod.rs, raw_vec.rs, const_ptr.rs, metad...
11.40	0.38	282 751	utils::huffman_new::Rolling...	simple_compress: huffman_new.rs
10.38	10.38	2 095 738	core::num::<impl u8>::pow	simple_compress: uint_macros.rs
8.87	1.53	4 224 310	core::iter::range::<impl cor...	simple_compress: range.rs
8.42	4.04	1 330 474	<core::iter::adapters::enu...	simple_compress: enumerate.rs, option.rs
8.00	0.77	668 448	core::slice::<impl [T]>::rot...	simple_compress: mod.rs, mut_ptr.rs

I watched this youtube video a while ago where people where trying to compute like the sieve of eratosthenes for prime numbers, and they were able to win the competition because they used constant expressions so that they could be done at compile time. I implemented a similar thing in mine with my reverse_huffman function, so that if the number was a length number, it would be able to be found with a simple lookup table.

100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.99	0.00	1	(below main)	simple_compress
99.99	0.00	1	__libc_start_main@@GLIBC…	libc.so.6: libc-start.c
99.99	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.99	0.00	1	main	simple_compress
99.99	0.00	1	std::rt::lang_start	simple_compress: rt.rs
99.99	0.00	1	std::rt::lang_start_internal	simple_compress: rt.rs, mod.rs, intrinsic…
99.98	0.00	1	std::rt::lang_start::{{closur…	simple_compress: rt.rs, process.rs, proc…
99.98	0.00	1	std::sys_common::backtrac…	simple_compress: backtrace.rs, hint.rs
99.98	0.00	1	core::ops::function::FnOnce…	simple_compress: function.rs
99.98	0.00	1	simple_compress::main	simple_compress: simple_compress.rs, r…
97.35	3.21	1	utils::huffman::compress	simple_compress: huffman.rs
82.62	1.44	1	utils::huffman_new::lz77_c…	simple_compress: huffman_new.rs
56.46	15.35	131 560	utils::huffman_new::find_m…	simple_compress: huffman_new.rs, utils.rs
17.66	3.86	1 063 125	utils::huffman_new::Rolling…	simple_compress: huffman_new.rs
14.54	3.13	130 209	utils::get_n_bits_reverse	simple_compress: utils.rs
14.36	0.29	125 598	utils::reverse_huffman	simple_compress: utils.rs
13.80	2.73	668 448	utils::huffman_new::Rolling…	simple_compress: huffman_new.rs
13.52	8.20	4 432 098	<alloc::vec::Vec<T,A> as c…	simple_compress: mod.rs, raw_vec.rs, c…
12.86	0.42	282 751	utils::huffman_new::Rolling…	simple_compress: huffman_new.rs
9.97	1.72	4 210 800	core::iter::range::<impl cor…	simple_compress: range.rs
9.50	4.55	1 330 474	<core::iter::adapters::enu…	simple_compress: enumerate.rs, option.rs
9.02	0.87	668 448	core::slice::<impl [T]>::rot…	simple_compress: mod.rs, mut_ptr.rs
8.25	6.86	4 210 800	<core::ops::range::Range<…	simple_compress: range.rs, cmp.rs
8.15	6.24	668 448	core::slice::rotate::ptr_rotate	simple_compress: rotate.rs, mut_ptr.rs, …

Even after all of that, I still wasn’t able to get it to be the same speed as miniz_oxides. They were able to compress Moby Dick 32 times in a bit over 1 second, while I took a little less than 3 seconds. I realized that the multithreading was only able to half the speed of my implementation. I assume it’s because it starts off with a new chunk every time, and therefore can’t take big leaps like the regular one can. I think some things that would be helpful are the compile time optimizations for the numbers less than 144.

Here are my final benchmarks:

```
compressed length 1644019
Valid compression of test2.txt

real    0m0.242s
user    0m0.468s
sys     0m0.109s
+ ./target/release/simple_compress test4.txt
compressed length 3287592
Valid compression of test4.txt

real    0m0.472s
user    0m0.947s
sys     0m0.291s
+ ./target/release/simple_compress test8.txt
compressed length 6573387
Valid compression of test8.txt

real    0m0.819s
user    0m1.548s
sys     0m0.533s
+ ./target/release/simple_compress test32.txt
compressed length 26302922
Valid compression of test32.txt

real    0m3.426s
user    0m6.739s
sys     0m2.363s


compressed length 1330051
Valid compression of test2.txt

real    0m0.100s
user    0m0.064s
sys     0m0.000s

compressed length 2659153
Valid compression of test4.txt

real    0m0.167s
```

user 0m0.105s
sys 0m0.010s

compressed length 5317390
Valid compression of test8.txt

real 0m0.343s
user 0m0.239s
sys 0m0.010s

compressed length 21266766
Valid compression of test32.txt

real 0m1.153s
user 0m0.866s
sys 0m0.031s

As a final test, I compared it with Gzip.

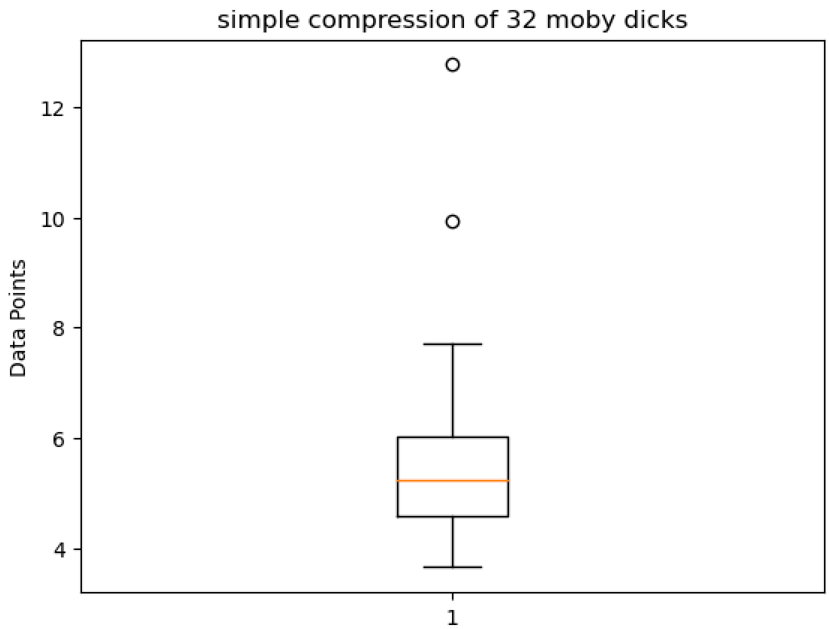
time gzip test32.txt

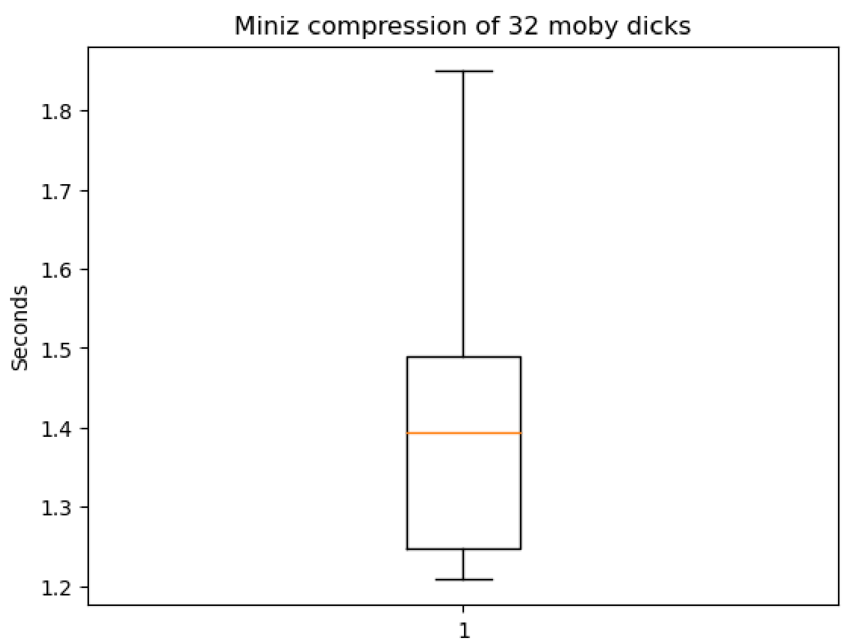
real 0m3.061s
user 0m2.091s
sys 0m0.040s

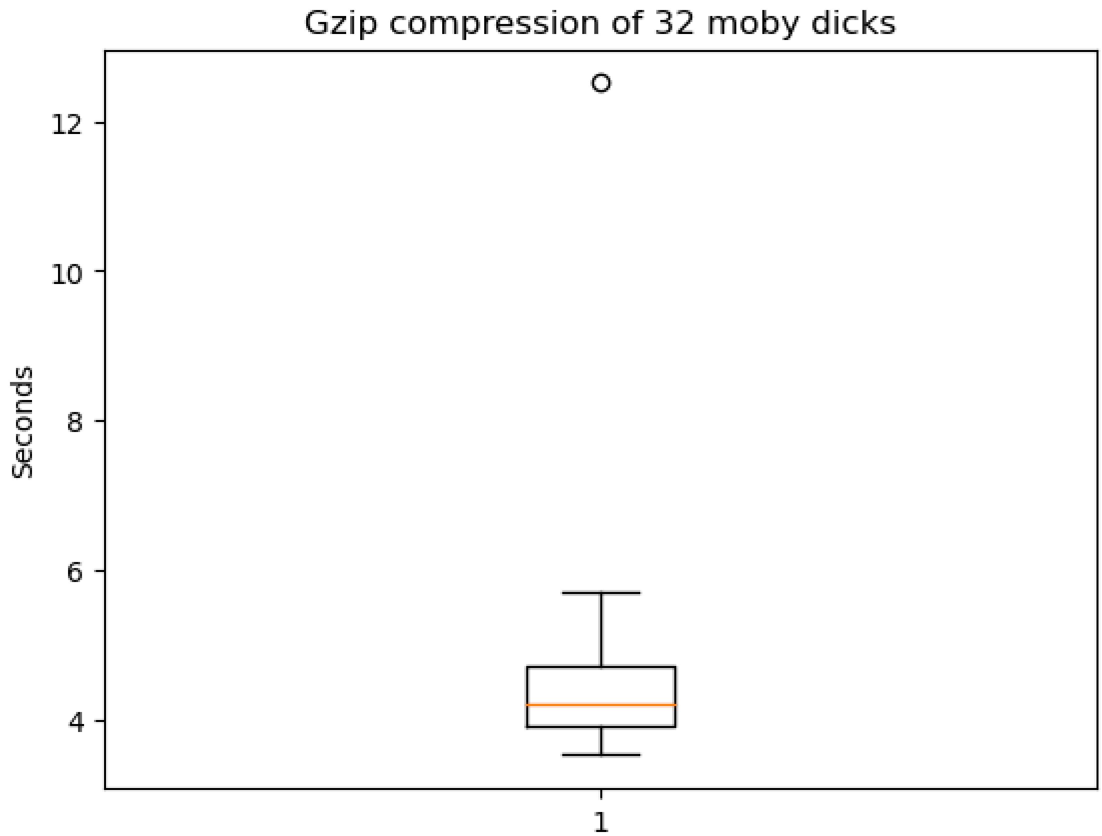
time gzip -d test32.txt.gz

real 0m0.615s
user 0m0.203s
sys 0m0.039s

Therefore, my implementation is around the same speed of gzip’s implementation, but both are substantially slower compared to minizoxides implementation, however, gzip has to also write the value to a file. I ran the script to compress and decompress moby dick 32 times. From the box plots, we can also see that my implementation is also a lot less consistent compared to miniz_oxides. Note that the numbers here are larger because I’m writing to a file instead of terminal


simple compression of 32 moby dicks
Data Points
12
10
8
6
4
1


Miniz compression of 32 moby dicks
Seconds
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1


Gzip compression of 32 moby dicks
Seconds
12
10
8
6
4
1