Executive Summary

The Arena Allocator Project aimed to evaluate different allocation algorithms for C libraries, specifically focusing on malloc and custom implementations using best fit, worst fit, next fit, and first fit strategies. Through benchmarking, I assessed the performance metrics, including allocation and freeing speeds. Our results indicate that certain implementations of algorithms consistently outperform other strategies across diverse workloads. I developed custom versions of malloc employing best fit, worst fit, next fit, and first fit algorithms, ensuring proper memory allocation, deallocation, and destruction of initialized memory blocks. I rigorously tested these implementations against a set of 20 diverse test cases provided by the professor, examining various allocation and deallocation scenarios. Following this, I designed a benchmarking program to assess the time complexity of my functions. This allowed me to compare the performance of my custom implementations against one another and evaluate how they fared in comparison to C's proprietary memory handling functions. I recorded the benchmarking data, created visual charts, and compiled a detailed report analyzing the outcomes.

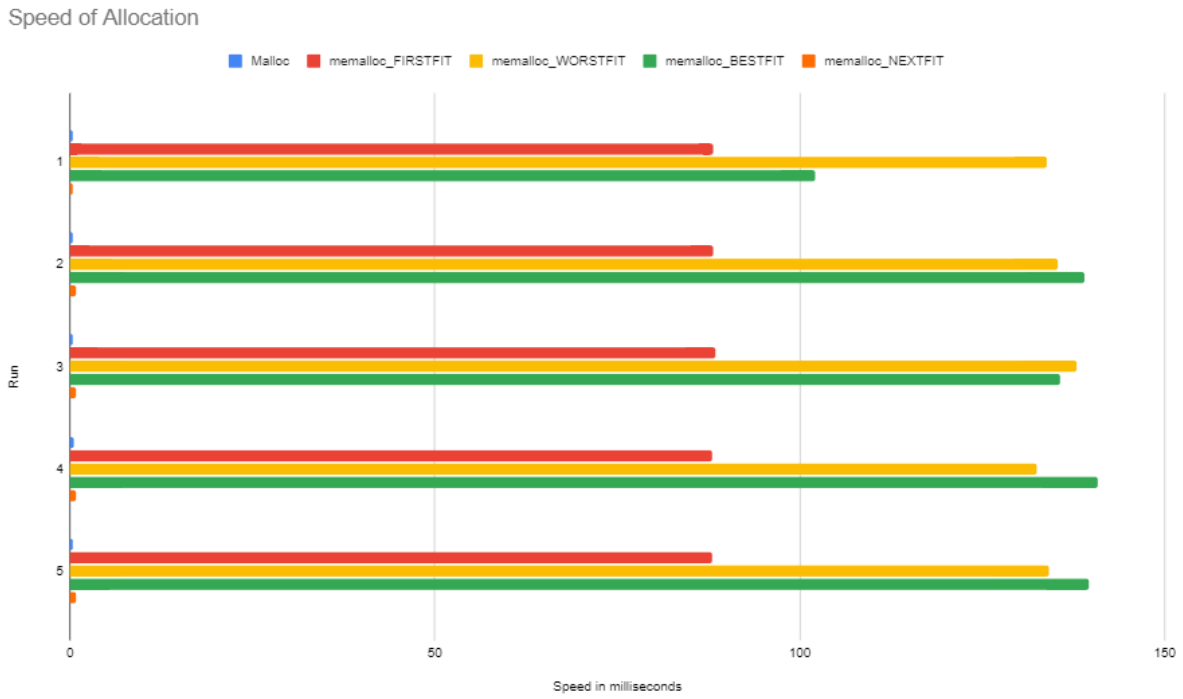
Data

The benchmarks were conducted to assess the time complexity of both the standard malloc function and custom implementations.. Clock times were recorded using the clock_t data type to mark the start and end times. A range of test cases and edge cases were considered, drawing from my knowledge and experiences gained by comparing my implementations against the test cases provided by the professor.

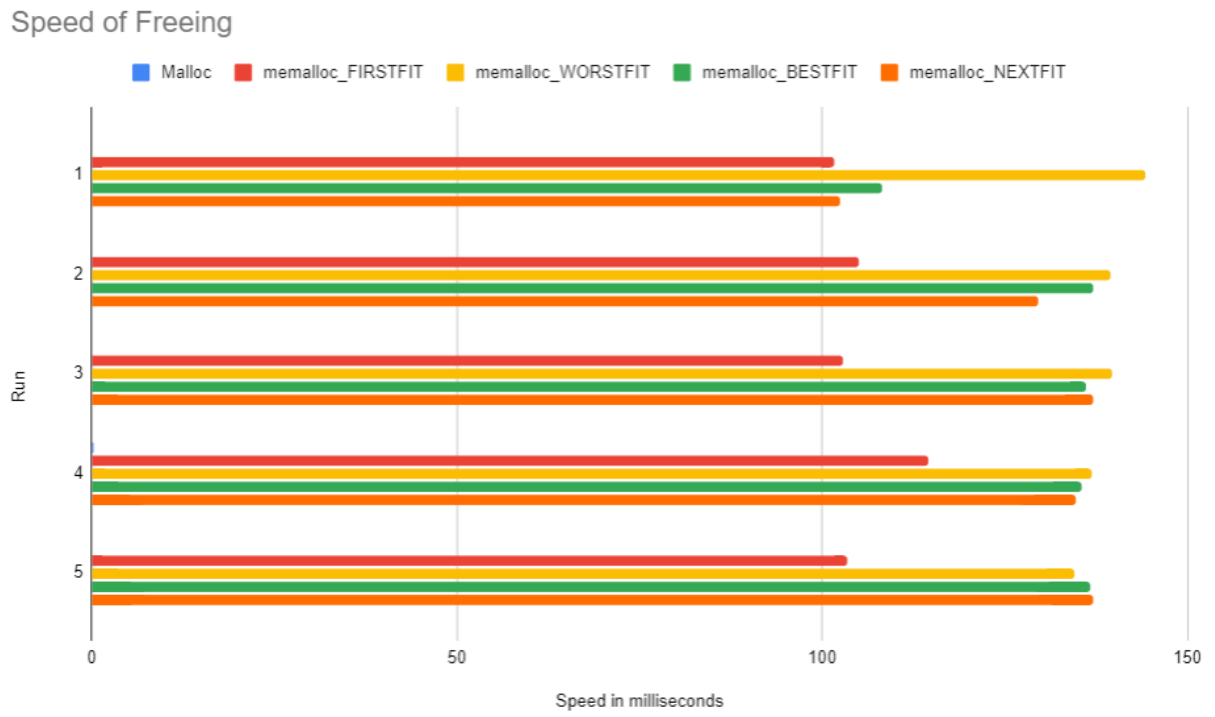
In total, I conducted five timed tests. The initial test involved initializing a block of memory and performing a malloc operation for each block. This process was repeated twice for each malloc implementation: once with a small number of allocations and small memory blocks,

and again with a large number of allocations and large memory blocks. Additionally, two free tests were carried out—one with large chunks and another with small chunks. Finally, the last test involved allocating and freeing a single block of memory 1,000 times.

The results were documented in a spreadsheet, and the collected data was utilized to create a bar graph, visually representing the outcomes of the tests.



Visualization of 5 runs of benchmark where malloc functions were called. The y axis represents the run number while the x axis represents the runtime. A lower runtime is a more efficient function.



Visualization of 5 runs of benchmark where free functions were called. The y axis represents the run number while the x axis represents the runtime in milliseconds. A lower runtime is a more efficient function.

Malloc

16 Byte Blocks Allocated 10000 times					
Run	1	2	3	4	5
Speed in millisec	0.335	0.31	0.309	0.541	0.409

Free 16 Byte Blocks 10000 times					
Run	1	2	3	4	5
Speed in millisec	0.108	0.102	0.115	0.149	0.107

4096 Byte Blocks Allocated 100000 times					
Run	1	2	3	4	5
Speed in millisec	157.601	162.389	155.892	162.971	177.391

Free 4096 Byte Blocks 100000 times					
Run	1	2	3	4	5
Speed in millisec	32.535	40.772	30.349	31.074	35.295

Allocate and free the same variable 1000 times					
Run	1	2	3	4	5
Speed in millisec	0.085	0.081	0.083	0.11	0.083

memalloc_FIRSTFIT

16 Byte Blocks Allocated 10000 times					
Run	1	2	3	4	5
Speed in millisec	88.098	88.186	88.363	87.969	87.941

memalloc_free 16 Byte Blocks 10000 times					
Run	1	2	3	4	5
Speed in millisec	101.545	105.1	102.826	114.544	103.338

4096 Byte Blocks Allocated 100000 times					
Run	1	2	3	4	5
Speed in millisec	8753.209	8955.448	8869.089	8861.133	8886.668

memalloc_free 4096 Byte Blocks 100000 times					
Run	1	2	3	4	5
Speed in millisec	40014.219	39990.823	40046.468	40053.641	44236.838

Allocate and memalloc_free the same variable 1000 times					
Run	1	2	3	4	5
Speed in millisec	0.024	0.024	0.048	0.047	0.051

memalloc_BESTFIT

16 Byte Blocks Allocated 10000 times					
Run	1	2	3	4	5
Speed in millisec	102.162	139.047	135.592	140.777	139.605

memalloc_free 16 Byte Blocks 10000 times					
Run	1	2	3	4	5
Speed in millisec	108.122	137.144	136.024	135.56	136.625

4096 Byte Blocks Allocated 100000 times					
Run	1	2	3	4	5
Speed in millisec	10923.032	13765.955	13782.679	13772.51	13755.305

memalloc_free 4096 Byte Blocks 100000 times					
Run	1	2	3	4	5
Speed in millisec	52191.549	52454.322	52478.323	52470.39	52034.248

Allocate and memalloc_free the same variable 1000 times					
Run	1	2	3	4	5
Speed in millisec	0.057	0.056	0.057	0.055	0.026

memalloc_NEXTFIT

16 Byte Blocks Allocated 10000 times					
Run	1	2	3	4	5
Speed in millisec	0.44	0.829	0.805	0.783	0.808

memalloc_free 16 Byte Blocks 10000 times					
Run	1	2	3	4	5
Speed in millisec	102.501	129.659	137.155	134.623	137.155

4096 Byte Blocks Allocated 100000 times					
Run	1	2	3	4	5
Speed in millisec	4.184	7.865	7.46	7.48	7.462

memalloc_free 4096 Byte Blocks 100000 times					
Run	1	2	3	4	5
Speed in millisec	49611.353	52564.055	52614.65	52566.367	52277.634

Allocate and memalloc_free the same variable 1000 times					
Run	1	2	3	4	5
Speed in millisec	0.056	0.055	0.055	0.076	0.026

memalloc_WORSTFIT

16 Byte Blocks Allocated 10000 times					
Run	1	2	3	4	5
Speed in millisec	133.893	135.373	137.959	132.488	134.067

memalloc_free 16 Byte Blocks 10000 times					
Run	1	2	3	4	5
Speed in millisec	144.291	139.546	139.679	136.852	134.457

4096 Byte Blocks Allocated 100000 times					
Run	1	2	3	4	5
Speed in millisec	12840.944	13552.481	13354.962	13347.891	13341.622

memalloc_free 4096 Byte Blocks 100000 times					
Run	1	2	3	4	5
Speed in millisec	52361.14	52355.159	52278.325	52205.874	51230.047

Allocate and memalloc_free the same variable 1000 times					
Run	1	2	3	4	5
Speed in millisec	0.048	0.056	0.044	0.056	0.026

Tables of each function's allocation/free benchmark results including the run number and the speed of each run in milliseconds.

```
@troy1eighty2 ➜ /workspaces/arena-allocator-troy1eighty2 (main) $ ./benchmark1

Little Allocations/ Little Blocks----------------------------------------------------------------
Function: malloc, Time taken: 0.343000 miliseconds for 10000 allocations with block size 16

Freeing Little Allocations/ Freeing Little Blocks------------------------------------------------
Function: free, Time taken: 0.136000 miliseconds for 10000 allocations with block size 16

Large Allocations/ Large Blocks------------------------------------------------------------------
Function: malloc, Time taken: 176.688000 miliseconds for 100000 allocations with block size 4096

Freeing Large Allocations/ Freeing Large Blocks--------------------------------------------------
Function: free, Time taken: 31.247000 miliseconds for 100000 allocations with block size 4096

Allocate, Free, Repeat 1000 times----------------------------------------------------------------
Function: free, Time taken: 0.081000 miliseconds for 1000 allocations/frees with block size 4096

@troy1eighty2 ➜ /workspaces/arena-allocator-troy1eighty2 (main) $ ./benchmark2

Little Allocations/ Little Blocks----------------------------------------------------------------
Function: memalloc_alloc, Time taken: 87.572000 miliseconds for 10000 allocations with block size 16

Freeing Little Allocations/ Freeing Little Blocks------------------------------------------------
Function: memalloc_free, Time taken: 101.395000 miliseconds for 10000 allocations with block size 16

Large Allocations/ Large Blocks------------------------------------------------------------------
Function: memalloc_alloc, Time taken: 8830.188000 miliseconds for 100000 allocations with block size 4096

Freeing Large Allocations/ Freeing Large Blocks--------------------------------------------------
Function: memalloc_free, Time taken: 39670.261000 miliseconds for 100000 allocations with block size 4096

Allocate, Free, Repeat 1000 times----------------------------------------------------------------
Function: memalloc_free, Time taken: 0.046000 miliseconds for 1000 allocations/frees with block size 4096

@troy1eighty2 ➜ /workspaces/arena-allocator-troy1eighty2 (main) $ ./benchmark3

Little Allocations/ Little Blocks----------------------------------------------------------------
Function: memalloc_alloc, Time taken: 91.902000 miliseconds for 10000 allocations with block size 16

Freeing Little Allocations/ Freeing Little Blocks------------------------------------------------
Function: memalloc_free, Time taken: 102.233000 miliseconds for 10000 allocations with block size 16

Large Allocations/ Large Blocks------------------------------------------------------------------
Function: memalloc_alloc, Time taken: 9833.029000 miliseconds for 100000 allocations with block size 4096

Freeing Large Allocations/ Freeing Large Blocks--------------------------------------------------
Function: memalloc_free, Time taken: 39686.540000 miliseconds for 100000 allocations with block size 4096

Allocate, Free, Repeat 1000 times----------------------------------------------------------------
Function: memalloc_free, Time taken: 0.040000 miliseconds for 1000 allocations/frees with block size 4096
```

Terminal after running all 5 benchmarks, 1 for each malloc function.

Conclusion

The allocation speed bar chart illustrates that C's malloc significantly outperforms all of my custom malloc implementations. According to the benchmark result tables, C's malloc completes tasks in a fraction of a millisecond, whereas my implementations take upwards of a

minute to finish the benchmarking test cases. Surprisingly, my custom NEXT_FIT implementation performs relatively well when compared to malloc, but the other three custom implementations take significantly longer.

FIRST_FIT performs consistently, allocating blocks of memory in about the same time across all test cases. On the other hand, the remaining implementations fall within a similar time range, approximately 50 milliseconds apart. Freeing memory exhibits a similar pattern to malloc, with C's free function being so swift that it almost doesn't register on the bar chart. Freeing memory with the memalloc_free function I built takes relatively the same amount of time across all allocation algorithms, excluding C’s proprietary functions.

In summary, these benchmarks conclusively demonstrate that C's malloc vastly outpaces our custom implementations of FIRST_FIT, NEXT_FIT, WORST_FIT, and BEST_FIT. Among the custom implementations, NEXT_FIT performs the best, while both BEST_FIT and WORST_FIT perform the poorest in terms of speed and efficiency.