Executive Summary:

In this project I developed a custom malloc implementation. This implementation is functional and will select the right addresses however, it is much slower than the standard malloc. Additionally, as detailed in this pdf, Next Fit performs very poorly which could be a result of a poor implementation of it.

Algorithms:

For this project, 4 different heap allocation algorithms were implemented to carry out malloc. These algorithms include:

- First Fit: Starts at the top of the heap and chooses the next available memory block big enough to hold the requested allocation.
- Next Fit: Initially starts at the top of the heap and finds the next available memory block. After the first time, Next Fit will start from where it last left off.
- Best Fit: Searches through the entire heap and finds the first smallest block of available memory big enough to hold the allocation.
- Worst Fit: Searches through the entire heap and finds the first largest block of available memory big enough to hold the allocation.

Test Implementation:

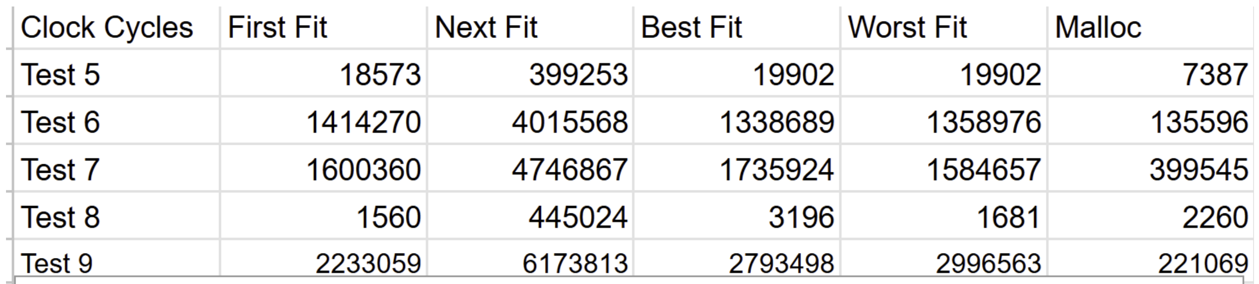
In order to benchmark the different allocation algorithms that were implemented against each other as well as the standard malloc, 5 additional tests were made. The tests behave as follows:

- Test 5: This test simulates memory fragmentation and examines how the different memory allocation strategies handle fragmented memory and free space. It does so by first allocating blocks of various sizes and then freeing nonadjacent blocks. More memory is then allocated.
- Test 6:This test emphasizes allocator performance and efficiency under repetitive, predictable workloads. Fragmentation, splitting, and coalescing are not factors, making it an ideal benchmark for base performance. It achieves this by allocating a large number of blocks and then freeing all of them repetitively.
- Test 7: This test creates a randomized memory allocation and deallocation pattern. It achieves this by allocating blocks of random sizes and then freeing half of those blocks. Then all blocks are freed and this process is repeated many times.
- Test 8: This test is designed to evaluate the allocator's performance and efficiency when handling large memory allocations. By repeatedly allocating and freeing a 1 MB block, it stresses the system's ability to manage heap growth and deallocation.
- Test 9: This test evaluates the allocator's ability to handle splitting and coalescing of memory blocks by creating fragmentation and repeatedly allocating and freeing memory of varying sizes. This test is an expansion of test 5 by freeing every other block and then

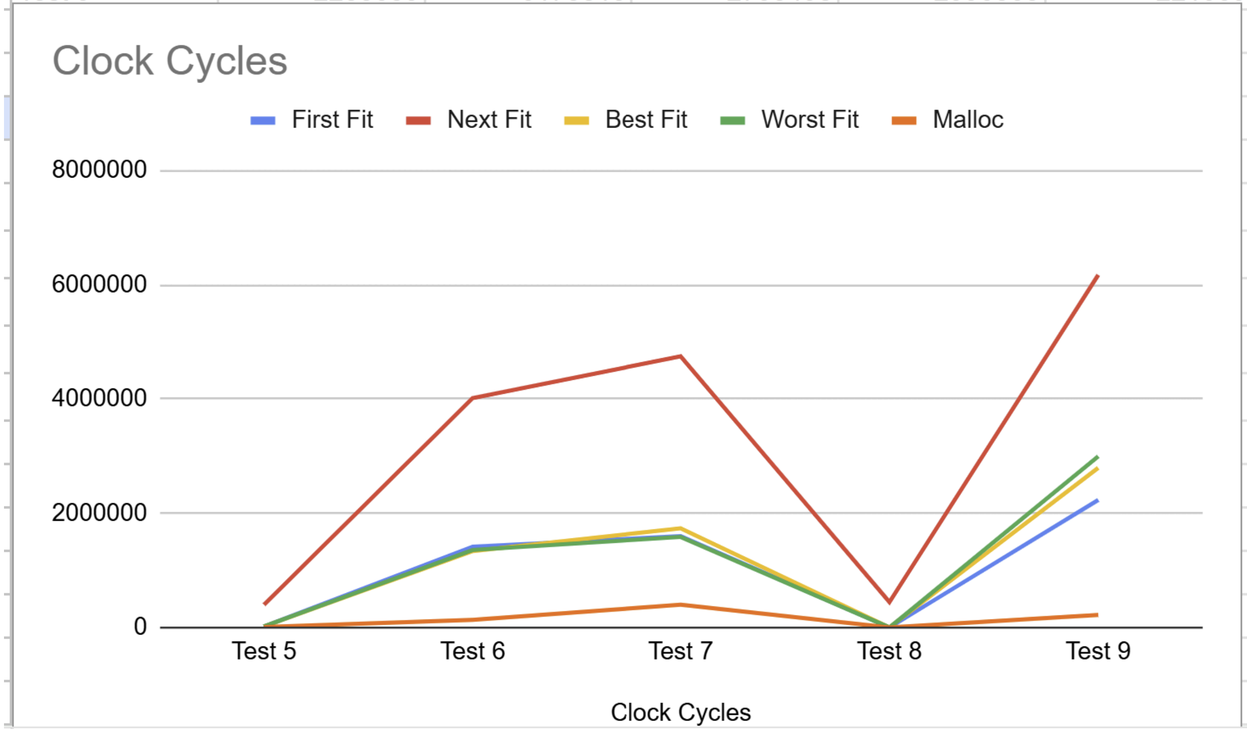
allocating blocks smaller than some of the freed blocks, forcing the allocator to split larger free blocks.

<u>Test Results:</u>

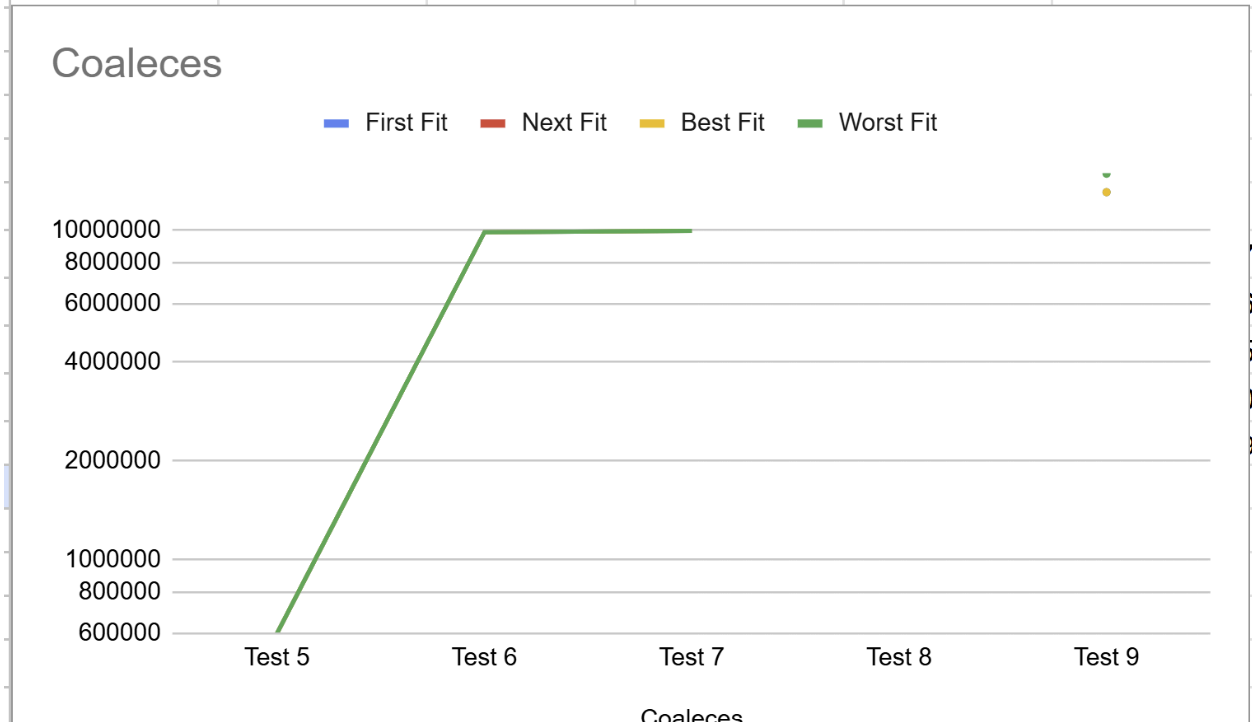
Below are images of the tables of data received from the tests as well as the respective graphs made from said data.



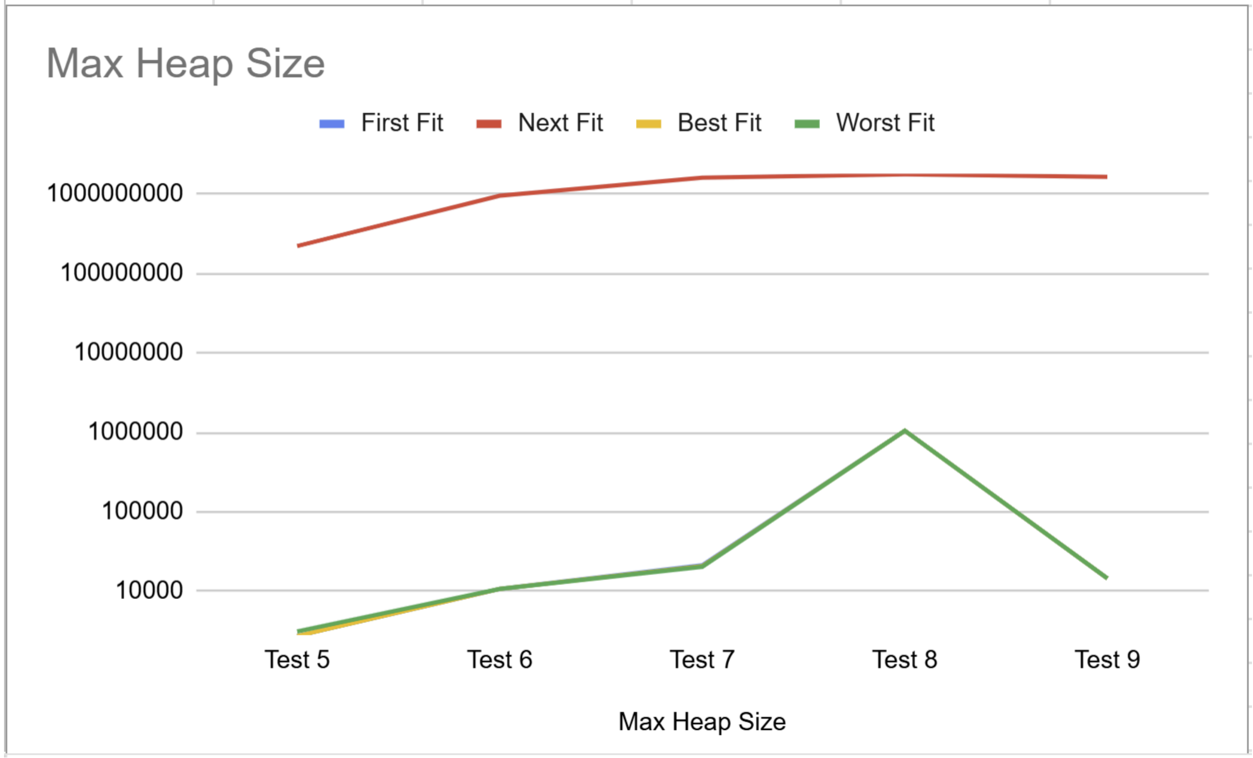
Clock Cycles	First Fit	Next Fit	Best Fit	Worst Fit	Malloc
Test 5	18573	399253	19902	19902	7387
Test 6	1414270	4015568	1338689	1358976	135596
Test 7	1600360	4746867	1735924	1584657	399545
Test 8	1560	445024	3196	1681	2260
Test 9	2233059	6173813	2793498	2996563	221069



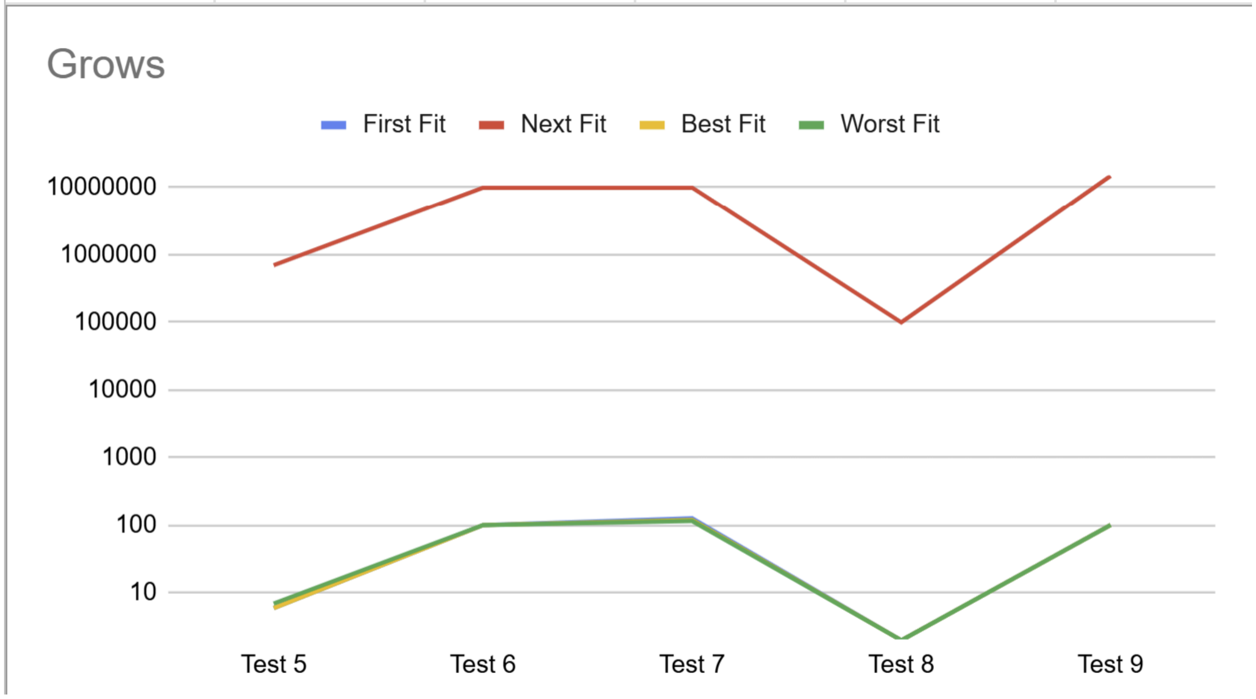
Coaleces	First Fit	Next Fit	Best Fit	Worst Fit
Test 5	600000	0	600000	600000
Test 6	9900000	0	9900000	9900000
Test 7	9999992	0	9999993	9999997
Test 8	0	0	0	0
Test 9	13100000	0	13100000	14900000



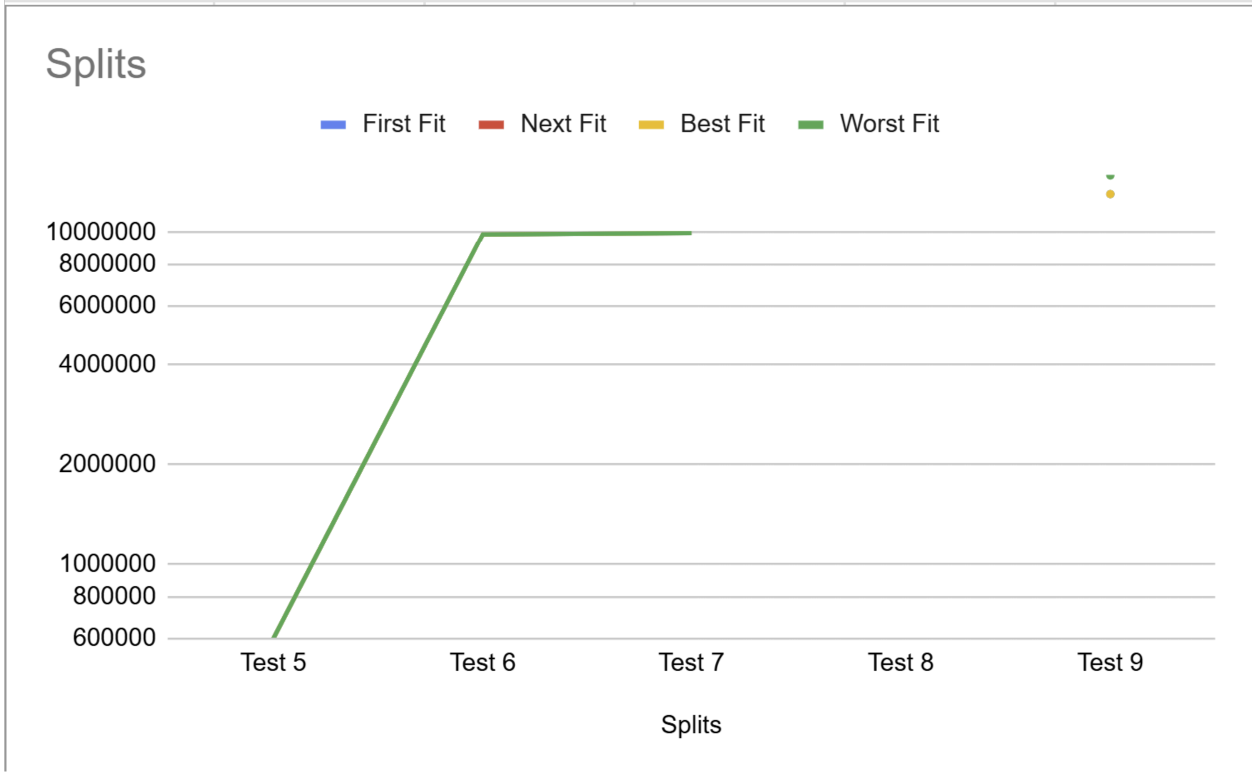
Max Heap Size	First Fit	Next Fit	Best Fit	Worst Fit
Test 5	2716	222801056	2716	3100
Test 6	10656	960001056	10656	10656
Test 7	21036	1620474836	20592	20232
Test 8	1049664	1781585952	1049664	1049664
Test 9	14456	1660001056	14456	14456



Grows	First Fit	Next Fit	Best Fit	Worst Fit
Test 5	6	700001	6	7
Test 6	101	10000001	101	101
Test 7	127	10000001	120	116
Test 8	2	100001	2	2
Test 9	101	15000001	101	101



Splits	First Fit	Next Fit	Best Fit	Worst Fit
Test 5	599996	0	599996	599995
Test 6	9899901	0	9899901	9899901
Test 7	9999867	0	9999875	9999883
Test 8	0	0	0	0
Test 9	13099901	0	13099901	14899901



Interpretation of Results:

Splits: Splitting occurs when a larger block is divided into smaller pieces to satisfy allocation requests.

- First Fit, Best Fit, and Worst Fit: These strategies produce comparable split counts, except for Worst Fit in Test 9, where it has substantially more splits.Worst Fit tends to allocate from the largest block, leaving many leftover smaller fragments, which increases splitting when those fragments are reused.
- Next Fit: Produces 0 splits across all tests because it grows the heap instead of splitting blocks, as evident from the Grows metric.

Summary:

Next Fit's reliance on heap growth avoids splitting but results in inefficient memory use (see Max Heap Size). Worst Fit suffers from excessive splits in fragmented scenarios like Test 9, highlighting its inefficiency in reuse.

Grows: Heap growth indicates the number of times additional memory is requested from the OS.

- First Fit, Best Fit, and Worst Fit: Show comparable, minimal heap growth in most tests, indicating efficient memory reuse.
- Next Fit: Requires extreme heap growth in all tests, reflecting its inability to reuse freed blocks efficiently. For example, in Test 7, Next Fit grew the heap 10,000,001 times, while other strategies kept growth minimal.

Summary: Next Fit's heap growth makes it inefficient for most scenarios, as it doesn't properly recycle freed memory.

Coalesces: Coalescing refers to merging adjacent free blocks into larger ones to reduce fragmentation.

- First Fit, Best Fit, and Worst Fit: Have high coalescing rates across tests, efficiently reclaiming free memory.
- Next Fit: Has 0 coalesces in all tests due to its tendency to grow the heap rather than reuse or merge existing blocks.

Summary: The high coalescing count for First Fit, Best Fit, and Worst Fit indicates their capability to manage fragmentation, unlike Next Fit.

Clock Cycles: Clock cycles measure the execution time of the tests and provide a sense of performance overhead.

- Next Fit: Performs poorly in terms of clock cycles due to excessive heap growth and lack of memory reuse. In Test 9, Next Fit took 6,173,813 clock cycles, compared to First Fit's 2,233,059 and Best Fit's 2,793,498.
- First Fit: Generally performs the fastest in most tests, balancing minimal splits and efficient coalescing.
- Best Fit and Worst Fit: Comparable to First Fit but slower due to the additional overhead of finding the smallest (Best Fit) or largest (Worst Fit) free block.

Summary: Standard Malloc is the most time-efficient overall, while Next Fit is significantly slower due to its poor reuse and growth strategy and the other custom allocations are slower.

Max Heap Size: The maximum heap size is the total memory allocated by the program.

- Next Fit: Requires excessive heap size in all tests due to its inability to reuse memory. For example, in Test 5, Next Fit's heap grew to 222,801,056 bytes, compared to 2,716 bytes for First Fit.
- First Fit, Best Fit, and Worst Fit: Maintain minimal heap size across tests, efficiently reusing memory.

Summary: Next Fit is highly inefficient in terms of memory usage, with massive heap size growth in all tests.

Conclusion on AI Performance:

1. Did the AI assistant help? Yes I would say that the ai assistant helped substantially as it wrote almost all of the code.
2. Did it hurt? Yes. While it wrote almost all of the code, it still took multiple days to finish the assignment. I would find that it would often get stuck in a loop where it recommended the same things over and over until I opened up a new chat on ChatGPT and restarted. Because of this, I wound up having to closely review and understand the entirety of the code anyway and I think the project took just as much time as it normally would have without AI.
3. Where did the AI tool excel? The AI tool excelled at creating Best Fit, Worst Fit, malloc, realloc, and calloc and it did so on the first try.
4. Where did it fail? The AI tool was especially bad at creating Next Fit and kept creating code that seg faulted for it. It was also especially terrible at debugging and was never able to find the issues. Even though Next Fit now works, its still the worst allocation algorithm by far in almost every test case.
5. Do you feel you learned more, less, or the same if you had implemented it fully on your own? I feel as though I learned around the same amount due to some of the faulty code it produces that caused me to have to debug the entire program and learn what everything was doing. Had it produced the code first try however, I definitely would have learned less.
6. If you leaned more, what was do you think you learned more of? N/A