Executive Summary

This report presents an analysis of memory allocation strategies implemented as part of my custom malloc system. Four allocation algorithms were tested: First Fit, Best Fit, Worst Fit, and Next Fit. The standard system malloc function was also used as a means to test the effectiveness of my custom malloc. Performance was measured using a benchmark test that simulated various memory patterns that allowed me to stress test my custom malloc and compare it against the system malloc, and the execution time was recorded for each allocator. Results demonstrated the trade-offs between fragmentation, allocation time, showing the effectiveness of different algorithms.

Description of the Algorithms Implemented

- First Fit: Scans the linked list from the start and selects the first available block that is large enough to satisfy the allocation request.
- Next Fit: Similar to First Fit, but continues the search from the last allocated block rather than starting from the beginning each time. Wraps around to the beginning if needed.
- Best Fit: Iterates through the entire list to find the smallest block that is large enough, in hopes of minimizing wasted space
- Worst Fit: Searches for the largest available block that can satisfy the request, under the assumption that splitting larger blocks will leave more useful fragments.
- System Malloc: The default memory allocator provided by the operating system, used as a benchmark to compare the performance of the custom strategies.

Test Implementation

The benchmark test allocates and frees memory in various patterns, taking inspiration from the given test1-test4 that we should test our initial test cases. I made a stress test named benchmark.c which I used to compare my 4 custom allocators to the system malloc. The test was conducted by going through the following:

- Allocating 1000 blocks of 256 bytes.
- Freeing every other block.
- Allocating and immediately freeing a large 256,000-byte block.
- Allocating 2000 tiny blocks of 8 bytes and then frees them.
- Freeing the remaining initially allocated blocks.

Execution time was recorded using clock() from the C standard library by including time.h. The test was run five times using my four custom allocators and the system malloc(), which can be observed on the following page:

Allocater	Execution Time (s)	Heap Management Statistics
System Malloc	0.0003 seconds	Can only test based on performance
First Fit	0.0065 seconds	mallocs: 3002 frees: 3001 reuses: 0 grows: 3002 splits: 0 coalesces: 2499 blocks: 503 requested: 532096 max heap: 604144
Next Fit	0.0045 seconds	mallocs: 3002 frees: 3001 reuses: 2000 grows: 1002 splits: 2000 coalesces: 2000 blocks: 1002 requested: 532096 max heap: 540144
Best Fit	0.0139 seconds	mallocs: 3002 frees: 3001 reuses: 2000 grows: 1002 splits: 1750 coalesces: 2750 blocks: 2 requested: 532096 max heap: 540144
Worst Fit	0.0518 seconds	mallocs: 3002 frees: 3001 reuses: 2000 grows: 1002 splits: 2000 coalesces: 3000 blocks: 2 requested: 532096 max heap: 540144

Explanation and Interpretation of Results

- System Malloc performed the best due to decades of optimization and low overhead.
- First Fit was fast due to early exits from the allocation loop but showed potential fragmentation. However, surprisingly First Fit did not have the fastest time from my custom allocators.
- Next Fit offered a good balance of speed and fragmentation control but was sensitive to the current state of the heap and pointer position. I fully expected the Next Fit allocator to be part of the top fastest time in this test, but to my surprise, it was the first fastest from all of my custom allocators. I believe it was the fastest due to it looking at the whole list instead of going through each time looking for an open spot.
- Best Fit aimed for space efficiency but incurred more overhead due to a full scan of the list, leading to longer execution times. From my initial understanding, I believed that the Best Fit Allocator would have the longest execution time due to its building for efficiency but from my new understanding of Worst Fit it makes a little bit of sense why it was faster than WF.
- Worst Fit was slower than Best Fit and sometimes led to more fragmentation, as leftover large blocks could not be reused efficiently. As I stated previously I expected Best Fit to be slower, but understanding that they, on the surface, do essentially the same thing I can understand why WF took longer. Due to WF trying to find the biggest spot and fragment the extra space it would take more time to go through its process, unlike BF which aims to limit the fragmentation.