Executive Summary

This report provides an extensive analysis of the ‘malloc’ assignment in which we learn to implement our own version of malloc and free. With these implementations, we must perform heap management in a user process as we learned in class. The important caveat within this assignment is the use of AI. We are allowed the assistance of GenAI to assist us in developing this assignment. Once we obtain this AI generated code, it’s our job to edit this code to pass 8 test programs which range from testing heap management schemes to testing other specific implementations within the program. Some important outcomes to note are based on the AI’s ability to provide useful and meaningful code. The code the AI tool provided lacked in-depth understanding of the program. The provided framework was a helpful starting point for the assignment, but it lacked proficiency when implementing functions that required a deeper understanding of the concepts applied. Besides the fact that the use of AI is frowned upon in an academic setting, it would not be wise to fully rely on AI generated code. It doesn’t always provide accurate results, and the user’s understanding of the topic may not always be improved when using it.

Description of the Algorithms Implemented

First Fit

The first fit algorithm selects the first available block of memory big enough to handle the requested allocation. It starts from the beginning of the list of memory blocks and stops as soon as it finds a block that can perform the job. This makes it fairly quick when the appropriate space is found early in the search. This algorithm was provided by the professor, but the ‘findFreeBlock’ function goes through the memory blocks stored in ‘heapList’, check if each block is free and big enough for the request. If a block that meets this criteria is found, it stops and uses it, making this fit test fairly efficient.

Best Fit

The best fit algorithm looks through the entire list of memory blocks to find the smallest block that is big enough to fit the requested allocation. Using this algorithm, it minimizes wasted space which in turn reduces fragmentation. In my implementation, the algorithm starts from the beginning of the list ‘heapList’ and keeps track of the smallest fitting block using ‘best_fit’, initially set to ‘NULL’, and ‘best_size’, set to ‘SIZE_MAX’. As it iterates through each block (‘curr’), it checks if a block is free (‘curr->free == true’), and if it’s large enough to satisfy the request. If the current block is a better fit than the current ‘best_fit’, that variable is updated to reflect that current block. After completing the traversal, the function assigns ‘best_fit’ to ‘curr’ and returns it for allocation, ensuring the block with the smallest amount of leftover memory is used, minimizing fragmentation.

Next Fit

The next fit algorithm is a slight variation of the first fit test. Instead of always starting at the beginning of the memory list, it picks up where the last allocation left off. This makes for a more balanced approach, as it tends to spread allocations throughout the memory instead of clustering them near the beginning. In my implementation, ‘findFreeBlock’, continues searching from where the last successful allocation (‘last_allocated’) left off. If a previous allocation exists (‘last_allocated != NULL’), the search begins from the next block (‘last_allocated->next’). The function continues traversing the list, checking if each list is free and large enough. If the test reaches the end of the list without finding an appropriate block, it starts from the beginning (‘curr = heapList’). The search ends when it finds a block that can accommodate the request, and ‘last_allocated’ is updated to this block. Subsequence allocations do not always start from the beginning of the list, which helps to spread memory usage more evenly and prevents clustering of allocations.

Worst Fit

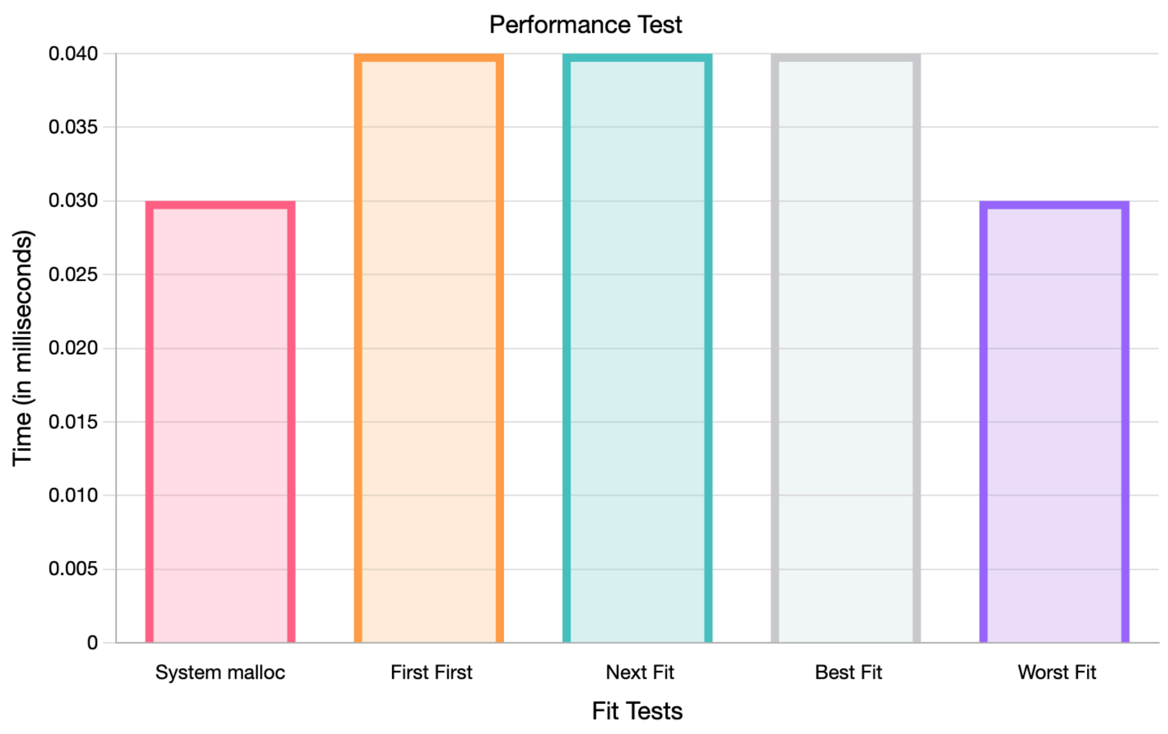
The worst fit algorithm in the implementation finds the largest free block available that can fulfill the allocation request. The ‘findFreeBlock’ function begins from ‘heapList’ and keeps track of the largest fitting block using ‘worst_fit’. As it traverses through the memory blocks, it checks if each block ‘curr’ is free, and can fit the requested allocation size (‘curr->free == true’ and ‘curr->size >= size’). If a block is found that’s larger than the current ‘worst_size’, ‘worst_fit’ is updated to point to that block. After it ends traversing the list, the function assigns ‘curr = worst_fit’ and returns it for allocation. Doing this, the largest available block is used, which can help maintain larger blocks of free memory for future use.

Test implementation

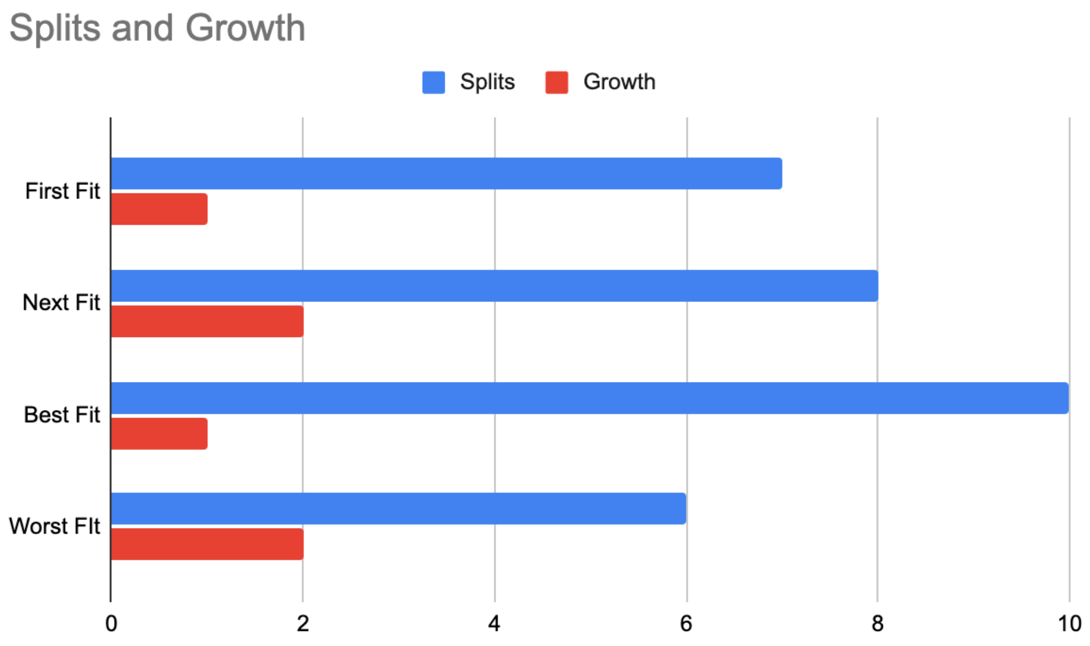
In benchmark.c, I evaluate the performance and behavior of our own memory allocator by running 5 tests: ‘basic_stress_test’, ‘random_allocation_test’, ‘sequential_growth_test’, ‘fragmentation_test’, and ‘reallocation_stress_test’. ‘Basic_stress_test’ allocates and frees a set number of blocks to see if allocation and deallocation work smoothly. The ‘random_allocation_rest’ uses blocks of random sizes, partially frees them, and reallocates others to see how the allocator deals with unpredictable memory use. The ‘sequential_growth_test’ checks if the allocator handles blocks that grow in size and then reset. The ‘fragmentation_test’ creates gaps by freeing every other block and continues to perform a large allocation to see if fragmentation affects memory usage. The ‘reallocation_stress_test’ resizes some of the initially allocated blocks to test the efficiency of reallocations. Each test tracks elapsed time to compare performance.

Test Results for All Five Candidates

Performance Test

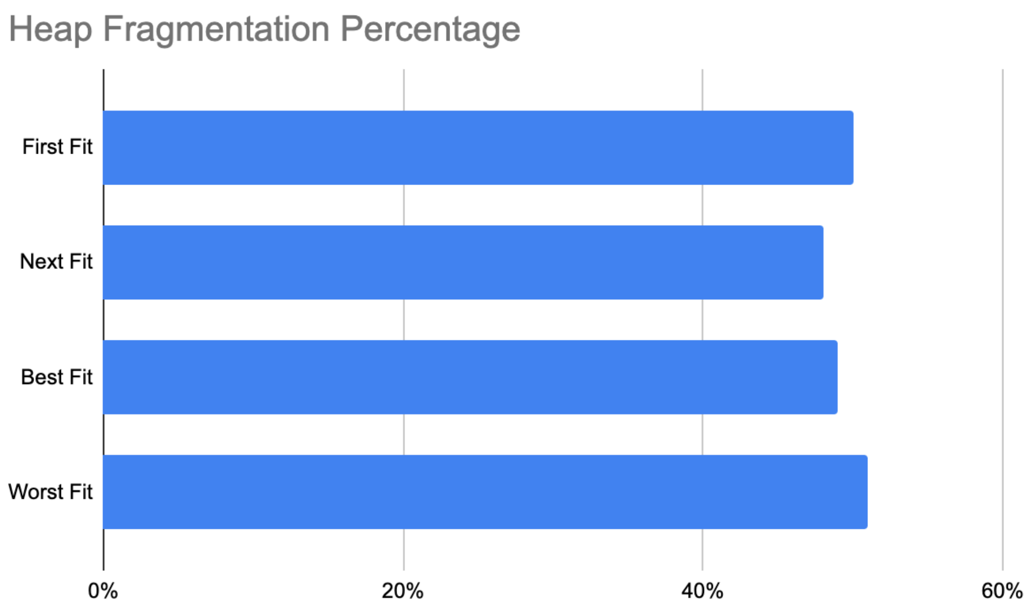


Splits and Heap Growth Test



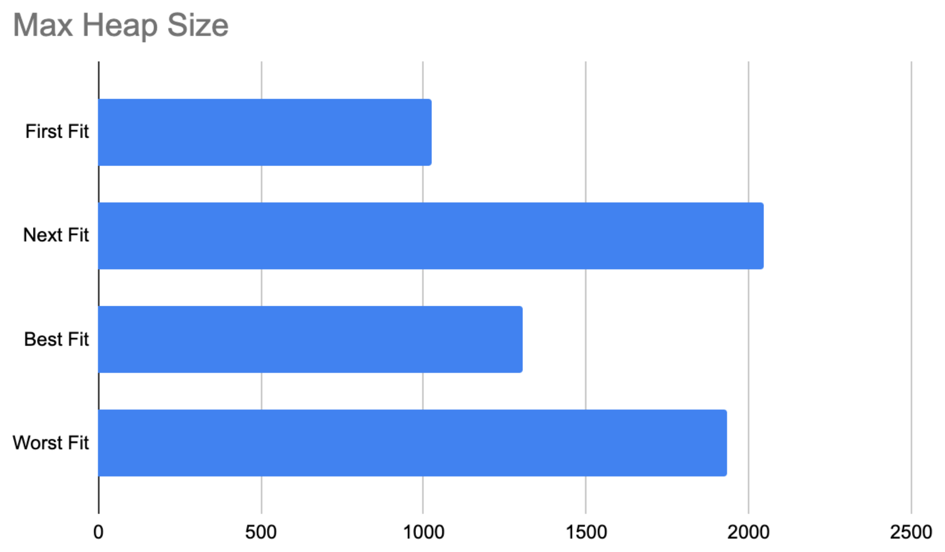
Disclaimer: My benchmark test program was not able to perform accurate split and growth results.

Heap Fragmentation Test



Disclaimer: My benchmark test was not able to provide accurate fragmentation percentages.

Max Heap Size



Disclaimer: Once again, my benchmark test was unable to provide accurate results.

Explanation and Interpretation of the Results Including Any Anomalies in the Test Results

Based on the test graphs, the system malloc generally outperformed custom strategies, with faster allocation times. First Fit and Next Fit had notably higher times, indicating slower performance. The splits and heap growth results showed frequent splits for all strategies, but heap growth measurements were inconsistent, as noted by the disclaimers I provided. Heap fragmentation appeared similar across all fit tests, suggesting issues in accurately capturing true fragmentation differences. In the max heap size test, Next Fit had the highest heap usage, indicating more frequent heap growth, likely due to inefficient reuse. Overall, inconsistencies and anomalies highlight limitations in the benchmark test's accuracy.

Conclusion On AI Performance

The AI tool’s assistance in this assignment fell short compared to the work of a student with a strong understanding of the concepts applied. It provided a helpful high-level overview of the program, but it struggled with functions that required a deeper understanding of concepts taught in class. It wouldn’t be in a user’s best interest to fully depend on the results that the tool provides; they should seek to develop a proficient understanding of the topic to be able to differentiate which parts of a program are viable and which aren’t. GenAI could be a valuable tool for getting started, but it should only serve as a tool, not something to replace the user's knowledge and understanding.

Did the AI assistant help?

The AI assistant did help with some functions. More specifically, it helped me gain a better understanding of some functions, and helped me get started on the assignment.

Did it hurt?

We are in college to learn, not just be able to memorize concepts like some robot, so I would say it did hurt to use help from an AI tool as the basis of this assignment. Yes you learn using AI tools, but it more so feels like you're learning how to create one assignment, and not like you're learning the concept to apply it somewhere else and develop that skill.

Where did the AI tool excel?

The AI tool gave me a great starting point towards the assignment. It was able to spit back a fully written out program based on the instructions given. It helped me gain an understanding of what the assignment was asking for, and filled in gaps with simple functions, allowing me to focus on more complex parts of the program that required more attention. Specifically, I was able to obtain nearly functional heap management code blocks on the first attempt, with only small adjustments needed for the next-fit test. The rest of the fit tests seemed to work from the first attempt.

Where did it fail?

In more conceptually heavy areas, the AI struggled to provide useful results. For functions like malloc and free, it failed to determine where to implement counters such as num_frees, num_grows, num_coalesces, and max_heap. The struggle to implement them caused some time consuming challenges, since resolving these issues required meticulous debugging by stepping through the code line by line in a debugger or having print statements throughout the program to identify where the counters were needed.

Do you feel you learned more, less, or the same if you had implemented it fully on your own?

I definitely feel I’ve learned less compared to implementing it myself. When you attempt to tackle a programming assignment that assigns you to create something from scratch, it can sound daunting. These types of assignments teach you how to think like a critical programmer, and not some person that only copies and pastes code. You have to figure out how to implement certain concepts in a way you’ve never had to before, and the result is rewarding and educational. In a class like this, every assignment pushes you to engage with specific, niche concepts, which ultimately build your skills to tackle broader, and more complex concepts.