İSTANBUL AYDIN UNIVERSITY

Faculty of Applied Sciences

Department of Management Information Systems

Final Project Questions

Operating Systems and Hardware

Instructor: Fatma Kuzey EDEŞ HUYAL

Course Code: UMI209 OPERATING SYSTEMS AND COMPUTER HARDWARE.

Student Name : **Raphael Aydın**

Student ID: B2481.011007

Due Date: January 7

Question 1- Explain Two File Allocation Methods (Bitmap vs Linked List) (20 Points)

A- Bitmap Allocation (10 points)

Explain the bitmap method by answering the following:

* How does a bitmap show which disk blocks or free or used?

Depending on what bit values (0 and 1) that disk blocks take and general system design rules show us disk blocks or free or used.

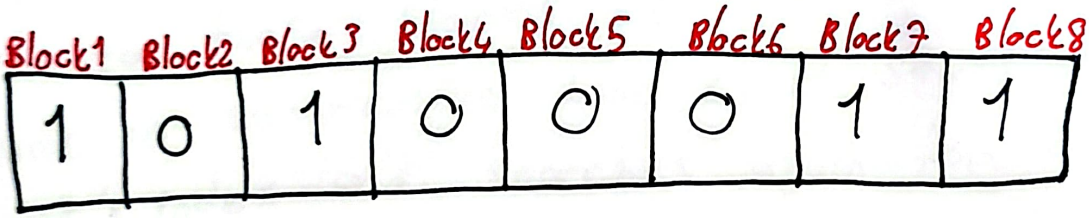


figure 1

figure 1 shows an image with 8 blocks; blocks with a value of 1 indicate a used disk blocks, while blocks with a value of 0 indicate an empty disk block.

& How does the operating system find a group of consecutive free blocks when a file needs space?

While there are many different methods and techniques, in its most basic form, if we are searching for suitable consecutive empty blocks for a file, the OS starts from the beginning of the bitmap or from where it left off if it has prior information. After finding the first 0 (empty) block, it starts a counter and then checks the following blocks. If the number of consecutive blocks is sufficent for our file, it writes the file to these blocks. If it encounters a 1 (allocated) block in consecutive blocks, it starts from the next block's index and searches again for blocks with suitable empty spaces.

Example

index 0 1 2 3 4 5 6 7 8 9 10

Bitmap : 1 1 1 1 0 0 1 1 0 0 0 0 0 1 1 1 0 .....

We need 3 free blocks

Bit 0 → 1 skip
Bit 1 → 1 skip
Bit 2 → 1 skip
Bit 3 → 1 skip
Bit 4 → 0 start counting and checks bit 5 and 6
Bit 5 → 0 counting continues
Bit 6 → 1 stop

It continues with the 7th bit and find sufficient space in the 8th, 9th, 10th bit

* How does the operating system free blocks a file is deleted?

when a file is deleted, the OS changes the block state in the bitmap from 1 to 0. However this doesn't mean it actually deletes the file. It also deletes the file's metadata. There is no physical deletion process on a disk. New files are physically written over the old data.

Example

Bitmap Segment = 1 0 1 1 0 0 1 1 0 1 - - - - - - -
Indexes = 0 1 2 3 4 5 6 7 8 9

If the file on the index 2 and 3 deleted uptaded bitmap segment will be = 1 0 0 0 0 0 1 1 0 1 - - - - - - -
Indexes = 0 1 2 3 4 5 6 7 8 9

* Give a small example using 8 blocks. Show the bitmap before and after an allocation.

I've given many demonstrations before, but I'll give another example.

Before Allocation

Indexes = 0 1 2 3 4 5 6 7
Bitmap = 1 0 1 1 1 0 0 1
Index 5 and 6 will be allocated

After Allocation

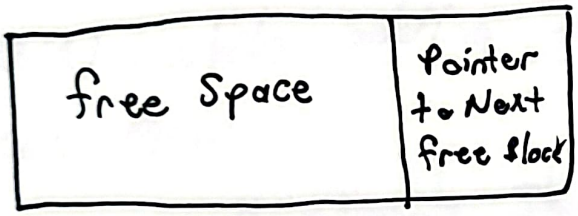
Indexes = 0 1 2 3 4 5 6 7
Bitmap = 1 0 1 1 1 1 1 1

B. Linked-List Allocation (10 points)

Explain the linked-list method by answering the following:

- How are free blocks organized when using a linked list?

Empty blocks have an empty space for files and a pointer that indicates the next empty block.



The system keeps the first empty block's pointer as the head.

- why does this Method NOT require consecutive blocks for a file?

Because the linked-List method is established via logical context (pointers), it allows for fragmented data storage on a disk. Each empty block can become part of a file. Therefore, consecutive empty blocks are not required.

- How does the system free all blocks of a file ?

The operating system finds and reads the blocks of the file, determining where it starts and where it ends. Then it releases the blocks one by one. After wards it adds the empty blocks to the free block's Linked-List one by one. This addition can be done using LIFO logic for adding to the head of the list or FIFO logic for adding to the tail of the list. If system adds free blocks to the beginning of the list Big-O will be O(1), while adding them to the end of the list makes O(n).

- Give a small example showing three blocks linked together.

file "ogrenciler.txt": Block0 → Block2 → Block5 → Eof

Fat Table free list: Block3 → Block7 → Block1 → Null

Index	Next
0	2 (ogrenciler.txt's part on block 0 points to 2)
1	free (in free list)
2	5 (ogrenciler.txt's part on blok 2 points to 5)
3	7 (free list head)
4	Eof (another file)
5	Eof (ogrenciler.txt's last block)
6	Eof (another file)
7	1 (free list continues)

Question 2 – Implement and Compare Two Allocation Algorithms (30 points)

Write your answers concisely and clearly. Complete all steps below.

A. Code Implementation (Using AI + Your Own Edits) (10 points)

- Use Gemini or ChatGPT to generate the first version of C code for (5 points):
  - Bitmap allocation
  - Linked-list allocation

  Make sure that the final code you submit is fully working. If you are not actively using C on your own computer, you may test and run your code using an online C compiler such as: OnlineGDB C Compiler .
- Use any programming language you feel comfortable working with and rewrite the generated code by applying (5 points):
  - Your own variable names
  - Your own formatting and structure
  - Detailed comments explaining each line
- Include both code versions in your submission:
  - Original AI-generated code
  - Your rewritten and commented code

B. Run Three Simple Experiments (10 points)

1. **Speed Test (100 allocations):**
   Run each algorithm 100 times. Measure how long it takes to allocate and free blocks. Which one is faster?
2. **Fragmentation Test:**
   Perform the same experiment for both allocators:
   - Make 20 random allocations of varying sizes.
   - Free exactly 5 of those allocations at random.

- Attempt to allocate one large block of size 12.

Report whether each method succeeds or fails, and explain the reason based on its allocation strategy.

3. **Allocation Trace:**

Perform the first **15** allocations using the **same sequence of allocation sizes** for both methods. For example, you may use the following fixed sequence:

$$2,\ 3,\ 5,\ 2,\ 4,\ 6,\ 1,\ 3,\ 5,\ 2,\ 4,\ 3,\ 2,\ 1,\ 5$$

After each allocation, print the full disk state (0 = free, 1 = allocated). Make sure the bitmap and linked-list allocators run with the **exact same** size sequence, so your comparison is meaningful.

Based on the 15 printed states, describe one clear difference in how the bitmap allocator and the linked-list allocator choose blocks on the disk.

AI GENERATED BITMAP IN C

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

#define BLOCK_COUNT 64
#define BITS_IN_BYTE 8
#define MAP_SIZE ((BLOCK_COUNT + BITS_IN_BYTE - 1) / BITS_IN_BYTE)

uint8_t allocation_map[MAP_SIZE];

void map_init() {
    memset(allocation_map, 0, MAP_SIZE);
}

int allocate_contiguous(int need) {
    if (need <= 0 || need > BLOCK_COUNT) return -1;

    int seq = 0;
    int begin = -1;

    for (int i = 0; i < BLOCK_COUNT; i++) {
        int by = i / BITS_IN_BYTE;
        int bi = i % BITS_IN_BYTE;

        if ((allocation_map[by] & (1 << bi)) == 0) {
            if (seq == 0) begin = i;
            seq++;

            if (seq == need) {
                for (int j = 0; j < need; j++) {
                    int blk = begin + j;
                    int b = blk / BITS_IN_BYTE;
                    int bt = blk % BITS_IN_BYTE;
                    allocation_map[b] |= (1 << bt);
                }
                return begin;
            }
        } else {
            seq = 0;
            begin = -1;
        }
    }
```

```
    return -1;
}

void free_contiguous(int start, int cnt) {
    for (int i = 0; i < cnt; i++) {
        int blk = start + i;
        int by = blk / BITS_IN_BYTE;
        int bi = blk % BITS_IN_BYTE;
        allocation_map[by] &= ~(1 << bi);
    }
}

void show_map() {
    for (int i = 0; i < BLOCK_COUNT; i++) {
        int by = i / BITS_IN_BYTE;
        int bi = i % BITS_IN_BYTE;
        printf("%d", (allocation_map[by] >> bi) & 1);
    }
    printf("\n");
}

void speed_test_bitmap() {
    printf("Bitmap Speed Test (100 iterations):\n");

    struct timeval begin_time, end_time;
    gettimeofday(&begin_time, NULL);

    for (int run = 0; run < 100; run++) {
        map_init();
        srand(run + 100);

        int allocs[100];
        int sizes[100];

        for (int i = 0; i < 100; i++) {
            sizes[i] = (rand() % 5) + 1;
            allocs[i] = allocate_contiguous(sizes[i]);
        }
         for (int i = 0; i < 100; i++) {
             if (allocs[i] != -1) {
                 free_contiguous(allocs[i], sizes[i]);
             }
         }
     }

     gettimeofday(&end_time, NULL);

     long start_us = begin_time.tv_sec * 1000000 + begin_time.tv_usec;
     long end_us = end_time.tv_sec * 1000000 + end_time.tv_usec;
     double elapsed = (end_us - start_us) / 1000000.0;

     printf("Time: %.6f seconds\n", elapsed);
     printf("Total allocations: 10,000\n");
     printf("Total frees: 10,000\n");
     printf("Operations/sec: %.0f\n\n", 20000.0 / elapsed);
 }

 void fragment_test_bitmap() {
     printf("Bitmap Fragmentation Test:\n");
     map_init();
     srand(555);

     int file_st[20];
     int file_sz[20];

     for (int i = 0; i < 20; i++) {
         file_sz[i] = (rand() % 5) + 1;
         file_st[i] = allocate_contiguous(file_sz[i]);
     }
```

```
        }
    }

    gettimeofday(&end_time, NULL);

    long start_us = begin_time.tv_sec * 1000000 + begin_time.tv_usec;
    long end_us = end_time.tv_sec * 1000000 + end_time.tv_usec;
    double elapsed = (end_us - start_us) / 1000000.0;

    printf("Time: %.6f seconds\n", elapsed);
    printf("Total allocations: 10,000\n");
    printf("Total frees: 10,000\n");
    printf("Operations/sec: %.0f\n\n", 20000.0 / elapsed);
}

void fragment_test_bitmap() {
    printf("Bitmap Fragmentation Test:\n");
    map_init();
    srand(555);

    int file_st[20];
    int file_sz[20];

    for (int i = 0; i < 20; i++) {
        file_sz[i] = (rand() % 5) + 1;
        file_st[i] = allocate_contiguous(file_sz[i]);
    }

    int free_idx[5];
    for (int i = 0; i < 5; i++) {
        free_idx[i] = rand() % 20;
        free_contiguous(file_st[free_idx[i]], file_sz[free_idx[i]]);
    }
    int big = allocate_contiguous(12);
    if (big != -1) {
        printf("Success: got 12 blocks at %d\n", big);
        free_contiguous(big, 12);
    } else {
        printf("Failed: no 12 contiguous blocks\n");
    }

    for (int i = 0; i < 20; i++) {
        int skip = 0;
        for (int j = 0; j < 5; j++) if (i == free_idx[j]) skip = 1;
        if (!skip && file_st[i] != -1) free_contiguous(file_st[i], file_sz[i]);
    }
    printf("\n");
}

void trace_test_bitmap() {
    printf("Bitmap Trace (15 steps):\n");
    map_init();

    int steps[] = {2,3,5,2,4,6,1,3,5,2,4,3,2,1,5};

    for (int s = 0; s < 15; s++) {
        allocate_contiguous(steps[s]);
        printf("Step %2d (%d): ", s+1, steps[s]);
        show_map();
    }
    printf("\n");
}

int main() {
    printf("=== BITMAP AI GENERATED ===\n\n");
    speed_test_bitmap();
    fragment_test_bitmap();
    trace_test_bitmap();
    return 0;
}
```

MY BITMAP CODE IN C

```c
// BITMAP ALLOCATOR -

// includes first
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
// timing stuff
#include <time.h>
#include <sys/time.h>

// here i defining our disk size
#define HOW_MANY_BLOCKS_WE_HAVE 64
// each byte has 8 bits
#define HOW_MANY_BITS_IN_ONE_BYTE 8
// calculate bitmap array size
#define BITMAP_ARRAY_NEEDED ((HOW_MANY_BLOCKS_WE_HAVE + HOW_MANY_BITS_IN_ONE_BYTE -
1) / HOW_MANY_BITS_IN_ONE_BYTE)

// global bitmap array - each bit = 1 block
uint8_t the_bitmap_thing[BITMAP_ARRAY_NEEDED];

// ------------------------------------------------------------------
// Function to start fresh
void start_bitmap_from_scratch() {
    // set all bytes to zero = all blocks free
    memset(the_bitmap_thing, 0, BITMAP_ARRAY_NEEDED);
    // in real system we'd log this maybe
}

// ------------------------------------------------------------------
// check if block is taken or free
int check_block_status(int block_num) {
    // which byte has our bit?
    int byte_position = block_num / HOW_MANY_BITS_IN_ONE_BYTE;
    // which bit inside that byte?
    int bit_offset = block_num % HOW_MANY_BITS_IN_ONE_BYTE;
    // get the bit value 0 or 1
    return (the_bitmap_thing[byte_position] >> bit_offset) & 1;
}

// ------------------------------------------------------------------
// MAIN ALLOCATION FUNCTION - finds consecutive blocks
int find_and_take_blocks(int how_many_we_want) {
    // basic error check
    if (how_many_we_want <= 0 || how_many_we_want > HOW_MANY_BLOCKS_WE_HAVE) {
        return -1;  // invalid request
    }

    int consecutive_counter = 0;      // how many free blocks in a row
    int maybe_start_here = -1;        // where we might start allocation

    // scan through all blocks 0 to 63
    for (int current_block = 0; current_block < HOW_MANY_BLOCKS_WE_HAVE;
current_block++) {
        // calculate byte and bit positions
        int which_byte = current_block / HOW_MANY_BITS_IN_ONE_BYTE;
        int which_bit = current_block % HOW_MANY_BITS_IN_ONE_BYTE;
```

```
        // check if this block is free (bit = 0)
        if ((the_bitmap_thing[which_byte] & (1 << which_bit)) == 0) {
            // it's free!
            if (consecutive_counter == 0) {
                maybe_start_here = current_block;  // mark as potential start
            }
            consecutive_counter++;  // increment our counter

            // did we find enough consecutive blocks?
            if (consecutive_counter == how_many_we_want) {
                // here  we found enough space
                // now mark all these blocks as taken (set bits to 1)
                for (int i = 0; i < how_many_we_want; i++) {
                    int block_to_mark = maybe_start_here + i;
                    int mark_byte = block_to_mark / HOW_MANY_BITS_IN_ONE_BYTE;
                    int mark_bit = block_to_mark % HOW_MANY_BITS_IN_ONE_BYTE;
                    // set the bit to 1 using OR operation
                    the_bitmap_thing[mark_byte] |= (1 << mark_bit);
                }
                return maybe_start_here;  // return starting block
            }
        } else {
            // block is taken - reset our search
            consecutive_counter = 0;
            maybe_start_here = -1;
        }
    }

    // if we get here, we didn't find enough consecutive blocks
    return -1;  // failure
}
```

```
// ------------------------------------------------------------------
// function to free blocks
void give_back_blocks(int start_block, int block_count) {
    // just clear the bits for each block
    for (int i = 0; i < block_count; i++) {
        int block_num = start_block + i;
        int byte_idx = block_num / HOW_MANY_BITS_IN_ONE_BYTE;
        int bit_pos = block_num % HOW_MANY_BITS_IN_ONE_BYTE;
        // clear bit to 0 using AND with complement
        the_bitmap_thing[byte_idx] &= ~(1 << bit_pos);
    }
}

// ------------------------------------------------------------------
// show current bitmap state
void show_current_bitmap() {
    for (int blk = 0; blk < HOW_MANY_BLOCKS_WE_HAVE; blk++) {
        printf("%d", check_block_status(blk));
    }
    printf("\n");
}

// ------------------------------------------------------------------
// TEST 1: SPEED TEST - run 100 times
void do_speed_test_bitmap_human() {
    printf(">>> TEST 1: BITMAP SPEED TEST <<<\n");
    printf("We'll run 100 complete cycles (allocate 100 + free 100 each time)\n");
    printf("That's 100 × 200 = 20,000 total operations\n\n");
```

```
    // high precision timer setup it will be explained by handwritte
    struct timeval time_start, time_end;
    gettimeofday(&time_start, NULL);  // start timing

    // main test loop - 100 iterations
    for (int iteration_num = 0; iteration_num < 100; iteration_num++) {
        // fresh start for each iteration
        start_bitmap_from_scratch();
        // use different seed each time but consistent
        srand(iteration_num * 123);

        // arrays to track what we allocated
        int allocation_starts[100];
        int allocation_sizes[100];

        // ---- ALLOCATION PHASE ----
        for (int i = 0; i < 100; i++) {
            // random size between 1 and 5 blocks
            allocation_sizes[i] = (rand() % 5) + 1;
            // try to allocate
            allocation_starts[i] = find_and_take_blocks(allocation_sizes[i]);
        }

        // ---- FREEING PHASE ----
        for (int i = 0; i < 100; i++) {
            if (allocation_starts[i] != -1) {  // if allocation succeeded
                give_back_blocks(allocation_starts[i], allocation_sizes[i]);
            }
        }
    }

    // end timing
    gettimeofday(&time_end, NULL);

    // calculate elapsed time in seconds
    long start_micro = time_start.tv_sec * 1000000 + time_start.tv_usec;
    long end_micro = time_end.tv_sec * 1000000 + time_end.tv_usec;
    double total_seconds = (end_micro - start_micro) / 1000000.0;
```

```
    // print results
    printf("RESULTS:\n");
    printf("Total time: %.6f seconds\n", total_seconds);
    printf("Total iterations: 100\n");
    printf("Operations per iteration: 200 (100 alloc + 100 free)\n");
    printf("TOTAL operations: 20,000\n");
    printf("Operations per second: %.0f ops/sec\n", 20000.0 / total_seconds);
    printf("Average time per operation: %.3f microseconds\n\n",
           (total_seconds * 1000000) / 20000.0);
}

// ------------------------------------------------------------------
// TEST 2: FRAGMENTATION TEST
void do_fragmentation_test_bitmap() {
    printf(">>> TEST 2: BITMAP FRAGMENTATION TEST <<<\n");

    start_bitmap_from_scratch();
    srand(999);  // fixed seed for reproducibility

    int file_start_positions[20];
    int file_block_counts[20];

    printf("Step 1: Creating 20 random files...\n");
    for (int file_id = 0; file_id < 20; file_id++) {
        // random file size 1-5 blocks
        file_block_counts[file_id] = (rand() % 5) + 1;
        file_start_positions[file_id] =
find_and_take_blocks(file_block_counts[file_id]);
        // printf("  File %d: %d blocks starting at %d\n",
        //        file_id+1, file_block_counts[file_id],
file_start_positions[file_id]);
    }
```

```
    printf("Step 2: Randomly selecting 5 files to delete...\n");
    int files_to_delete[5];
    for (int d = 0; d < 5; d++) {
        files_to_delete[d] = rand() % 20;
        printf("  Deleting file %d (frees up %d blocks)\n",
               files_to_delete[d]+1, file_block_counts[files_to_delete[d]]);
        give_back_blocks(file_start_positions[files_to_delete[d]],
                         file_block_counts[files_to_delete[d]]);
    }

    printf("Step 3: Trying to allocate a large file (12 blocks)...\n");
    int big_file_start = find_and_take_blocks(12);

    if (big_file_start != -1) {
        printf("  Succesfull Large file allocated starting at block %d\n",
big_file_start);
        give_back_blocks(big_file_start, 12);
    } else {
        printf("  Failed Cannot find 12 consecutive free blocks.\n");

    }

    // cleanup - free remaining files
    for (int f = 0; f < 20; f++) {
        int already_freed = 0;
        for (int d = 0; d < 5; d++) {
            if (f == files_to_delete[d]) already_freed = 1;
        }
        if (!already_freed && file_start_positions[f] != -1) {
            give_back_blocks(file_start_positions[f], file_block_counts[f]);
        }
    }
    printf("\n");
}
// ------------------------------------------------------------------
// TEST 3: ALLOCATION TRACE
void do_allocation_trace_bitmap() {
    printf(">>> TEST 3: BITMAP ALLOCATION TRACE <<<\n");
    printf("Allocation sequence: 2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5\n\n");

    start_bitmap_from_scratch();

    // the fixed sequence from the assignment
    int allocation_sequence[15] = {2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5};

    for (int step_number = 0; step_number < 15; step_number++) {
        find_and_take_blocks(allocation_sequence[step_number]);
        printf("After step %2d (allocate %d): ", step_number+1,
allocation_sequence[step_number]);
        show_current_bitmap();
    }


}
// MAIN FUNCTION
int main() {

    // run all three tests
    do_speed_test_bitmap_human();
    do_fragmentation_test_bitmap();
    do_allocation_trace_bitmap();

    printf("=== END ===\n");
    return 0;
}
```

AI GENERATED LINKED-LIST ALLOCATION IN C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

#define MAX_BLOCKS 64
#define NO_BLOCK -1

typedef struct {
    char info[30];
    int nxt;
} BlockNode;

BlockNode storage[MAX_BLOCKS];
int free_start = NO_BLOCK;

void setup() {
    for (int idx = 0; idx < MAX_BLOCKS; idx++) {
        if (idx < MAX_BLOCKS - 1) {
            storage[idx].nxt = idx + 1;
        } else {
            storage[idx].nxt = NO_BLOCK;
        }
    }
    free_start = 0;
}

int get_one_block() {
    if (free_start == NO_BLOCK) return NO_BLOCK;

    int taken = free_start;
    free_start = storage[free_start].nxt;
    storage[taken].nxt = NO_BLOCK;
    return taken;
}
int get_blocks(int num) {
    if (num <= 0) return NO_BLOCK;

    int first = NO_BLOCK;
    int prev = NO_BLOCK;

    for (int cnt = 0; cnt < num; cnt++) {
        int newb = get_one_block();
        if (newb == NO_BLOCK) {
            while (first != NO_BLOCK) {
                int nxt = storage[first].nxt;
                storage[first].nxt = free_start;
                free_start = first;
                first = nxt;
            }
            return NO_BLOCK;
        }

        if (prev != NO_BLOCK) {
            storage[prev].nxt = newb;
        } else {
            first = newb;
        }
        prev = newb;
    }
    return first;
}
```

```
void release_blocks(int begin) {
    if (begin == NO_BLOCK) return;

    int cur = begin;
    int last = begin;
    while (storage[cur].nxt != NO_BLOCK) {
        last = cur;
        cur = storage[cur].nxt;
    }

    storage[last].nxt = free_start;
    free_start = begin;
}

void display_state() {
    int status[MAX_BLOCKS] = {0};

    int cur = free_start;
    while (cur != NO_BLOCK) {
        status[cur] = 0;
        cur = storage[cur].nxt;
    }

    for (int i = 0; i < MAX_BLOCKS; i++) {
        if (status[i] == 0) {
            int isfree = 0;
            cur = free_start;
            while (cur != NO_BLOCK) {
                if (cur == i) {
                    isfree = 1;
                    break;
                }
                cur = storage[cur].nxt;
            }
            printf("%d", isfree ? 0 : 1);
        }
    }
    printf("\n");
}
```

```
void speed_test_list() {
    printf("Linked-List Speed Test (100 iterations):\n");

    struct timeval t1, t2;
    gettimeofday(&t1, NULL);

    for (int iter = 0; iter < 100; iter++) {
        setup();
        srand(iter + 200);

        int allocs[100];
        int sizes[100];

        for (int i = 0; i < 100; i++) {
            sizes[i] = (rand() % 5) + 1;
            allocs[i] = get_blocks(sizes[i]);
        }

        for (int i = 0; i < 100; i++) {
            if (allocs[i] != NO_BLOCK) {
                release_blocks(allocs[i]);
            }
        }
    }
```

```
    gettimeofday(&t2, NULL);

    long start_u = t1.tv_sec * 1000000 + t1.tv_usec;
    long end_u = t2.tv_sec * 1000000 + t2.tv_usec;
    double total_t = (end_u - start_u) / 1000000.0;

    printf("Time: %.6f seconds\n", total_t);
    printf("Total allocations: 10,000\n");
    printf("Total frees: 10,000\n");
    printf("Operations/sec: %.0f\n\n", 20000.0 / total_t);
}

void fragment_test_list() {
    printf("Linked-List Fragmentation Test:\n");
    setup();
    srand(777);

    int starts[20];
    int counts[20];

    for (int i = 0; i < 20; i++) {
        counts[i] = (rand() % 5) + 1;
        starts[i] = get_blocks(counts[i]);
    }

    int freed[5];
    for (int i = 0; i < 5; i++) {
        freed[i] = rand() % 20;
        release_blocks(starts[freed[i]]);
    }

    int big = get_blocks(12);
    if (big != NO_BLOCK) {
        printf("Success: got 12 blocks at %d\n", big);
        release_blocks(big);
    } else {
        printf("Failed\n");
    }

    for (int i = 0; i < 20; i++) {
        int skipit = 0;
        for (int j = 0; j < 5; j++) if (i == freed[j]) skipit = 1;
        if (!skipit && starts[i] != NO_BLOCK) release_blocks(starts[i]);
    }
    printf("\n");
}

void trace_test_list() {
    printf("Linked-List Trace (15 steps):\n");
    setup();

    int seq[] = {2,3,5,2,4,6,1,3,5,2,4,3,2,1,5};

    for (int s = 0; s < 15; s++) {
        get_blocks(seq[s]);
        printf("Step %2d (%d): ", s+1, seq[s]);
        display_state();
    }
    printf("\n");
}

int main() {

    speed_test_list();
    fragment_test_list();
    trace_test_list();
    return 0;
}
```

MY LINKED-LIST ALLOCATION CODE IN C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>  // for better timing

// constants
#define TOTAL_BLOCKS_IN_DISK 64
#define END_MARKER -1  // means "no next block"

// each block structure
typedef struct disk_block_node {
    char some_data[32];      // would be file data in real system
    int pointer_to_next;     // next block in chain
} BlockNode;

// global variables
BlockNode my_disk[TOTAL_BLOCKS_IN_DISK];  // the whole disk
int first_free_block = END_MARKER;         // head of free list

// ---------------------------------------------------------------
// setup function - initializes everything
void initialize_linked_system() {
    // link all blocks into one big free list
    for (int i = 0; i < TOTAL_BLOCKS_IN_DISK; i++) {
        // put some  data
        snprintf(my_disk[i].some_data, 32, "block-%d-initial", i);

        // chain them: 0 -> 1 -> 2 -> ... -> END_MARKER
        if (i < TOTAL_BLOCKS_IN_DISK - 1) {
            my_disk[i].pointer_to_next = i + 1;
        } else {
            my_disk[i].pointer_to_next = END_MARKER;  // last one
        }
    }
    // free list starts at block 0
    first_free_block = 0;
}
```

```c
// allocate ONE block
int grab_one_free_block() {
    if (first_free_block == END_MARKER) {
        return END_MARKER;  // no free blocks!
    }

    // take from front of free list LIFO style
    int block_we_took = first_free_block;

    // update free list head
    first_free_block = my_disk[first_free_block].pointer_to_next;

    // clear the next pointer in taken block
    my_disk[block_we_took].pointer_to_next = END_MARKER;

    return block_we_took;
}
```

```c
// allocate MULTIPLE blocks for a file
int allocate_file_blocks(int how_many_needed) {
    // check input
    if (how_many_needed <= 0) {
        return END_MARKER;
    }

    int first_block_of_file = END_MARKER;
    int previous_block = END_MARKER;

    // try to get each block we need
    for (int i = 0; i < how_many_needed; i++) {
        int new_block = grab_one_free_block();

        // check if we ran out of space
        if (new_block == END_MARKER) {
            //  out of space
            // need to free any blocks we already got
            while (first_block_of_file != END_MARKER) {
                int next_in_chain = my_disk[first_block_of_file].pointer_to_next;
                my_disk[first_block_of_file].pointer_to_next = first_free_block;
                first_free_block = first_block_of_file;
                first_block_of_file = next_in_chain;
            }
            return END_MARKER;  // fail
        }

        // put some data in the block
        snprintf(my_disk[new_block].some_data, 32, "file-data-%d", i);

        // link it into our file chain
        if (previous_block != END_MARKER) {
            my_disk[previous_block].pointer_to_next = new_block;
        } else {
            first_block_of_file = new_block;  // this is first block
        }
        previous_block = new_block;
    }

    return first_block_of_file;
}
```

```c
// free a file (all its blocks)
void free_up_file_blocks(int file_start_block) {
    if (file_start_block == END_MARKER) {
        return;  // nothing to free
    }

    // find the last block in this file's chain
    int current = file_start_block;
    int last_block = file_start_block;
    while (my_disk[current].pointer_to_next != END_MARKER) {
        last_block = current;
        current = my_disk[current].pointer_to_next;
    }

    // add entire chain to front of free list
    my_disk[last_block].pointer_to_next = first_free_block;
    first_free_block = file_start_block;
}
```

```c
// display current state (similar to bitmap output)
void show_linked_list_state() {
    // mark all blocks as "allocated" (1) initially
    int block_status[TOTAL_BLOCKS_IN_DISK];
    for (int i = 0; i < TOTAL_BLOCKS_IN_DISK; i++) {
        block_status[i] = 1;  // assuming allocated
    }

    // mark blocks in free list as free (0)
    int curr = first_free_block;
    while (curr != END_MARKER) {
        block_status[curr] = 0;  // free
        curr = my_disk[curr].pointer_to_next;
    }

    // print the status
    for (int blk = 0; blk < TOTAL_BLOCKS_IN_DISK; blk++) {
        printf("%d", block_status[blk]);
    }
    printf("\n");
}

// ------------------------------------------------------------------
// TEST 1: SPEED TEST (100 iterations)
void run_speed_test_linked_human() {
    printf(">>> TEST 1: LINKED-LIST SPEED TEST <<<\n");
    printf("Running 100 complete cycles (100 alloc + 100 free each time)\n");
    printf("Total: 100 × 200 = 20,000 operations\n\n");

    // high precision timing
    struct timeval begin_time, finish_time;
    gettimeofday(&begin_time, NULL);

    // main test loop - 100 iterations
    for (int iteration = 0; iteration < 100; iteration++) {
        // fresh start each iteration
        initialize_linked_system();
        // consistent but varying seed
        srand(iteration * 456);

        // track allocations
        int file_starts[100];
        int file sizes[100];
      // ---- ALLOCATION PHASE ----
      for (int i = 0; i < 100; i++) {
          file_sizes[i] = (rand() % 5) + 1;  // 1-5 blocks
          file_starts[i] = allocate_file_blocks(file_sizes[i]);
      }

      // ---- FREEING PHASE ----
      for (int i = 0; i < 100; i++) {
          if (file_starts[i] != END_MARKER) {
              free_up_file_blocks(file_starts[i]);
          }
      }
  }

  // end timing
  gettimeofday(&finish_time, NULL);

  // calculate time
  long start_usec = begin_time.tv_sec * 1000000 + begin_time.tv_usec;
  long end_usec = finish_time.tv_sec * 1000000 + finish_time.tv_usec;
  double total_sec = (end_usec - start_usec) / 1000000.0;
```

```
    // results
    printf("RESULTS:\n");
    printf("Total time: %.6f seconds\n", total_sec);
    printf("Iterations completed: 100\n");
    printf("Operations per iteration: 200\n");
    printf("TOTAL operations: 20,000\n");
    printf("Operations per second: %.0f ops/sec\n", 20000.0 / total_sec);
    printf("Average time per operation: %.3f microseconds\n\n",
           (total_sec * 1000000) / 20000.0);
}

// ------------------------------------------------------------
// TEST 2: FRAGMENTATION TEST
void run_fragmentation_test_linked() {
    printf(">>> TEST 2: LINKED-LIST FRAGMENTATION TEST <<<\n");

    initialize_linked_system();
    srand(888);  // fixed seed

    int file_beginnings[20];
    int how_many_blocks[20];

    printf("Step 1: Creating 20 random files...\n");
    for (int f = 0; f < 20; f++) {
        how_many_blocks[f] = (rand() % 5) + 1;  // 1-5 blocks
        file_beginnings[f] = allocate_file_blocks(how_many_blocks[f]);
        // printf("  File %d: %d blocks, starts at %d\n",
        //        f+1, how_many_blocks[f], file_beginnings[f]);
    }

    printf("Step 2: Randomly deleting 5 files...\n");
    int deleted_files[5];
    for (int d = 0; d < 5; d++) {
        deleted_files[d] = rand() % 20;
        printf("  Deleting file %d\n", deleted_files[d]+1);
        free_up_file_blocks(file_beginnings[deleted_files[d]]);
    }

    printf("Step 3: Trying to allocate large file (12 blocks)...\n");
    int big_file_start = allocate_file_blocks(12);


    if (big_file_start != END_MARKER) {
        printf("  Successfull 12-block file allocated starting at %d\n",
big_file_start);

        free_up_file_blocks(big_file_start);
    } else {
        printf("  Failed \n");
    }

    // cleanup
    for (int f = 0; f < 20; f++) {
        int was_deleted = 0;
        for (int d = 0; d < 5; d++) {
            if (f == deleted_files[d]) was_deleted = 1;
        }
        if (!was_deleted && file_beginnings[f] != END_MARKER) {
            free_up_file_blocks(file_beginnings[f]);
        }
    }
    printf("\n");
}
```

```c
// TEST 3: ALLOCATION TRACE
void run_allocation_trace_linked() {
    printf(">>> TEST 3: LINKED-LIST ALLOCATION TRACE <<<\n");
    printf("Same sequence: 2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5\n\n");

    initialize_linked_system();

    int the_sequence[15] = {2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5};

    for (int step = 0; step < 15; step++) {
        allocate_file_blocks(the_sequence[step]);
        printf("Step %2d (allocate %d): ", step+1, the_sequence[step]);
        show_linked_list_state();
    }


}

// ------------------------------------------------------------------
// MAIN
int main() {


    run_speed_test_linked_human();
    run_fragmentation_test_linked();
    run_allocation_trace_linked();

    printf("END\n");
    return 0;
}
```

TEST RESULTS

```
>>> TEST 1: BITMAP SPEED TEST <<<
We'll run 100 complete cycles (allocate 100 + free 100 each time)
That's 100 × 200 = 20,000 total operations

RESULTS:
Total time: 0.000785 seconds
Total iterations: 100
Operations per iteration: 200 (100 alloc + 100 free)
TOTAL operations: 20,000
Operations per second: 25477707 ops/sec
Average time per operation: 0.039 microseconds

>>> TEST 2: BITMAP FRAGMENTATION TEST <<<
Step 1: Creating 20 random files...
Step 2: Randomly selecting 5 files to delete...
  Deleting file 14 (frees up 4 blocks)
  Deleting file 13 (frees up 3 blocks)
  Deleting file 4 (frees up 2 blocks)
  Deleting file 7 (frees up 3 blocks)
  Deleting file 17 (frees up 1 blocks)
Step 3: Trying to allocate a large file (12 blocks)...
  FAILED! Cannot find 12 consecutive free blocks.

>>> TEST 3: BITMAP ALLOCATION TRACE <<<
Allocation sequence: 2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5

After step  1 (allocate 2): 11000000000000000000000000000000000000000000000000000000000000000000000000000000
After step  2 (allocate 3): 11111000000000000000000000000000000000000000000000000000000000000000000000000000
After step  3 (allocate 5): 11111111110000000000000000000000000000000000000000000000000000000000000000000000
After step  4 (allocate 2): 11111111111110000000000000000000000000000000000000000000000000000000000000000000
After step  5 (allocate 4): 11111111111111110000000000000000000000000000000000000000000000000000000000000000
After step  6 (allocate 6): 11111111111111111111110000000000000000000000000000000000000000000000000000000000
After step  7 (allocate 1): 11111111111111111111111000000000000000000000000000000000000000000000000000000000
After step  8 (allocate 3): 11111111111111111111111110000000000000000000000000000000000000000000000000000000
After step  9 (allocate 5): 11111111111111111111111111111110000000000000000000000000000000000000000000000000
After step 10 (allocate 2): 11111111111111111111111111111111000000000000000000000000000000000000000000000000
After step 11 (allocate 4): 11111111111111111111111111111111111100000000000000000000000000000000000000000000
After step 12 (allocate 3): 11111111111111111111111111111111111111000000000000000000000000000000000000000000
After step 13 (allocate 2): 11111111111111111111111111111111111111110000000000000000000000000000000000000000
After step 14 (allocate 1): 11111111111111111111111111111111111111111000000000000000000000000000000000000000
After step 15 (allocate 5): 11111111111111111111111111111111111111111111000000000000000000000000000000000000
=== END OF BITMAP TESTS ===
```

```
>>> TEST 1: LINKED-LIST SPEED TEST <<<
Running 100 complete cycles (100 alloc + 100 free each time)
Total: 100 × 200 = 20,000 operations

RESULTS:
Total time: 0.001160 seconds
Iterations completed: 100
Operations per iteration: 200
TOTAL operations: 20,000
Operations per second: 17241379 ops/sec
Average time per operation: 0.058 microseconds

>>> TEST 2: LINKED-LIST FRAGMENTATION TEST <<<
Step 1: Creating 20 random files...
Step 2: Randomly deleting 5 files...
  Deleting file 4
  Deleting file 3
  Deleting file 6
  Deleting file 7
  Deleting file 20
Step 3: Trying to allocate large file (12 blocks)...
  SUCCESS! 12-block file allocated starting at 57

>>> TEST 3: LINKED-LIST ALLOCATION TRACE <<<
Same sequence: 2, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 3, 2, 1, 5

Step  1 (allocate 2): 11000000000000000000000000000000000000000000000000000000000000
Step  2 (allocate 3): 11111000000000000000000000000000000000000000000000000000000000
Step  3 (allocate 5): 11111111110000000000000000000000000000000000000000000000000000
Step  4 (allocate 2): 11111111111100000000000000000000000000000000000000000000000000
Step  5 (allocate 4): 11111111111111100000000000000000000000000000000000000000000000
Step  6 (allocate 6): 11111111111111111111100000000000000000000000000000000000000000
Step  7 (allocate 1): 11111111111111111111111000000000000000000000000000000000000000
Step  8 (allocate 3): 11111111111111111111111111000000000000000000000000000000000000
Step  9 (allocate 5): 11111111111111111111111111111100000000000000000000000000000000
Step 10 (allocate 2): 11111111111111111111111111111111000000000000000000000000000000
Step 11 (allocate 4): 11111111111111111111111111111111111100000000000000000000000000
Step 12 (allocate 3): 11111111111111111111111111111111111111100000000000000000000000
Step 13 (allocate 2): 11111111111111111111111111111111111111111000000000000000000000
Step 14 (allocate 1): 11111111111111111111111111111111111111111110000000000000000000
Step 15 (allocate 5): 11111111111111111111111111111111111111111111111100000000000000
=== END OF LINKED-LIST TESTS ===
```

1) Speed Test (100 Allocation)

Run Each algorithm 100 times. Measure how long it takes to allocate and free blocks. Which one is faster?

According to test result, when we ran 100 allocation and 100 free for 100 times, it took 0.000785 seconds with the bitmap method and 0.001160 seconds with the linked-list allocation method. Normally, linked list method is much faster, but because the test size is small, the operation was completed much faster when we use bitmap method because bitmap is on cache but linked-list method not. If we raise the disk up to 1024 blocks not 64 bitmap going to 1.10times slower than Linked-List method. If we look up even longer disks 8192 blocks Linked-List going to be 12 times faster.

Report whether each method succeeds or fails and explain the reason based on its allocation strategy.

In the fragmentation test, after making 20 allocations and free 5 blocks, bitmap method failed to place 12 blocks of data and raised an error. This was because it couldn't find 12 consecutive free blocks. However it work without error using the linked-list method because the linked-list method doesn't work by finding consecutive free blocks. All free blocks can become part of the file when we use Linked-List method

C. Short Reflection

- What you wrote yourself ?

I made variable names more descriptive. I added detailed comments to each function. I wrote the test functions in more detail. I solved the time mesaurement problem. When I used the "clock()" structure, the time was 0.0000 in results and I couldn't measure the duration accurately. So, to get more precise mesaurements, I added the "gettimeofday()" structure to both the AI-generated code and my own code. This solved the problem of the process duration constantly showing as 0.

- What the AI generated ?

for bitmap allocation;
Various functions and structures like simple free function clearing bits. Array structure using uint8_t

for Linked-List Allocation
Disk block structure with embedded next pointer.
free list management using LIFO allocation.
Basic file chain creation and deletion.

- What you chaged why ?

Entire time calculation process
LIFO logic for linked-list allocation
Testing results detail
Comments for entire code blocks
Upgrade the test functions.

I do those changes because when it's ai-generated due to block size I can not get the results how I want. The time specifically always showed 0 and I can not make test's the code can not runnable I added test functions and redesign the code blocks to be able to run code on every platforms.

What you learned from writing and testing the algorithms ?

Due to time complexity perspective which is O(1) for linked-list allocation method and O(n) for bitmap allocation method the result was surprised me. It is not what I expected because I was thinking in theoretically. for small discs where bitmap fits in CPU cache, bitmap can outperform linked-list despite worse theoretical complexity.

Bitmap seeks contiguity it is better for sequential performance but suffers from external fragmentation. Otherside linked-list avoids fragmentation.

The CPU cache size, discs size and access patterns make differencies along both method.

for benchmarking the highy-resolution time structures must be used because the process time is under 10^{-4} seconds.

Question 3 – Memory Allocation with Linked Lists (40 points)

In this question, you will implement and compare three classic memory allocation algorithms that operate on a **linked list of free memory blocks**. The three allocation strategies you will study are:

- Best Fit
- Worst Fit
- Next Fit

Assume the system manages a simulated memory of **100 units** (indexed 0–99), represented as a linked list of free segments of the form `[start, length]`.
You will complete the following tasks:

A. Explanation of the Linked-List Allocation Algorithms (10 points)

Explain how each allocation algorithm works when searching the linked list of free blocks. In your explanation, describe the behavior of:

- **Best Fit:** how it selects the smallest block that is large enough, how the chosen block is allocated, and what happens to the leftover space.

The best fit algorithm aims to find the block that best matches the desired size. It first scans the entire list of free blocks. If the block size and the desired size match perfectly, it's called a perfect match, and the block is removed from the free list. If there isn't a perfect match and the block size is larger, it allocates a portion of the free space to the file, and the remaining space is stays for future request in the free list.

This approach leaves space available for files that may require larger storage space in the future. However leftover fragments going to be so small they can not satisfy future request.

- **Worst Fit:** how it selects the largest available block, how the chosen block is allocated, and how the remaining free memory is handled.

The worse fit algorithm upon receiving an allocation request, it traverse through the entire list of free blocks. and compare their sizes to find the largest block. It then test the suitability of that block for request. Afterward, it performs the allocation and leaves a leftover space. The purpose of this is to keep the leftover space available for future use. However, its disadvantage is that because the system fills large blocks first, it cannot handle future large-scale operations, leading to external fragmentation

- **Next Fit:** how it begins searching from the most recent allocation position, how it chooses a block, and what happens to any unallocated space after the request is served.

The next-fit algorithm is an improved version of the first-fit algorithm. It doesn't scan the entire list of free blocks from the beginning, it remembers the address of the last allocated block. When a new request arrives it continues from where it left off, traversing the list and compare the block sizes. Once it finds a suitable block it allocates it. If the found block size is larger than the request size, it leaves a leftover, which becomes ~~the~~ a new free block. Then it moves the pointer to the next block, which becomes the starting point for the new request.

B. Implementation (15 points)

Use any programming language you feel comfortable working with and write code that implements the following functions:

- `allocate_best_fit(size)`
- `allocate_worst_fit(size)`
- `allocate_next_fit(size)`
- `free(start, size)`

Each algorithm should:

- search the free list according to its strategy,
- allocate a block and split the free segment,
- update the free-list linked structure,
- merge adjacent free blocks when freeing.

Include comments explaining how each algorithm chooses a block.

Python Code

```python
"""
 Best Fit, Worst Fit, Next Fit Implementations
"""

import random
import time


# ==================== CORE DATA STRUCTURES ====================
class FreeBlock:
    """Represents a free memory block in the linked list"""

    def __init__(self, start, size):
        self.start = start  # Starting address of the free block
        self.size = size  # Size of the free block
        self.next = None  # Pointer to next free block


class MemoryAllocator:
    """Main memory allocator implementing all three algorithms"""

    def __init__(self, total_memory=100):
        """Initialize allocator with total memory size"""
        self.total_memory = total_memory
        # Initially one big free block covering entire memory
        self.free_list = FreeBlock(0, total_memory)
        # Track allocated blocks for freeing (block_id -> (start, size))
        self.allocated_blocks = {}
        # Roving pointer for Next Fit algorithm
        self.next_fit_ptr = self.free_list
        # Counter for generating unique block IDs
        self.block_id_counter = 0
        # For tracking allocation sequence for experiments
        self.allocation_sequence = []

    # ==================== UTILITY METHODS ====================

    def print_free_list(self, algorithm=""):
        """Print the current free list in readable format"""
        print(f"{algorithm:10s} Free List: ", end="")
        current = self.free_list
        while current:
            print(f"[{current.start:3d}-{current.start + current.size - 1:3d}]({cur
rent.size:3d})", end="")
            if current.next:
                print(" → ", end="")
            current = current.next
        print()

    def coalesce_free_blocks(self):
        """Merge adjacent free blocks in the free list"""
        if not self.free_list:
```

```
            return

        current = self.free_list
        while current and current.next:
            # If current block ends where next block starts, merge them
            if current.start + current.size == current.next.start:
                current.size += current.next.size
                current.next = current.next.next
            else:
                current = current.next

    def reset(self):
        """Reset allocator to initial state"""
        self.free_list = FreeBlock(0, self.total_memory)
        self.allocated_blocks = {}
        self.next_fit_ptr = self.free_list
        self.block_id_counter = 0
        self.allocation_sequence = []

    # ==================== ALLOCATION ALGORITHMS ====================

    def allocate_best_fit(self, size, block_id=None):
        """
        BEST FIT: Find smallest free block that fits the request
        Scans entire list to minimize leftover space
        """
        if block_id is None:
            block_id = self.block_id_counter
            self.block_id_counter += 1

        if not self.free_list:
            return None  # No free memory

        best_block = None
        best_prev = None
        current = self.free_list
        prev = None

        # Search entire list for the best (smallest) fit
        while current:
            if current.size >= size:
                # Check if this is better than current best
                if best_block is None or current.size < best_block.size:
                    best_block = current
                    best_prev = prev
            prev = current
            current = current.next

        if not best_block:
            return None  # No block large enough

        # Allocate from the best block
        allocated_start = best_block.start

        # Update the free list
```

```
        if best_block.size == size:  # Exact fit - remove entire block
            if best_prev:
                best_prev.next = best_block.next
            else:
                self.free_list = best_block.next
        else:  # Split the block - allocate from beginning
            best_block.start += size
            best_block.size -= size

        # Record the allocation
        self.allocated_blocks[block_id] = (allocated_start, size)
        self.allocation_sequence.append(('allocate', block_id, size, 'best_fit'))

        return allocated_start

    def allocate_worst_fit(self, size, block_id=None):
        """
        WORST FIT: Find largest free block available
        Always picks biggest block to maximize leftover space
        """
        if block_id is None:
            block_id = self.block_id_counter
            self.block_id_counter += 1

        if not self.free_list:
            return None

        worst_block = None
        worst_prev = None
        current = self.free_list
        prev = None

        # Search entire list for the worst (largest) fit
        while current:
            if current.size >= size:
                if worst_block is None or current.size > worst_block.size:
                    worst_block = current
                    worst_prev = prev
            prev = current
            current = current.next

        if not worst_block:
            return None

        # Allocate from the worst (largest) block
        allocated_start = worst_block.start

        # Update the free list
        if worst_block.size == size:  # Exact fit
            if worst_prev:
                worst_prev.next = worst_block.next
            else:
                self.free_list = worst_block.next
        else:  # Split the block
            worst_block.start += size
```

```
            worst_block.size -= size

        self.allocated_blocks[block_id] = (allocated_start, size)
        self.allocation_sequence.append(('allocate', block_id, size, 'worst_fit'))

        return allocated_start

    def allocate_next_fit(self, size, block_id=None):
        """
        NEXT FIT: Start search from last allocation position
        Uses roving pointer to distribute allocations across memory
        """
        if block_id is None:
            block_id = self.block_id_counter
            self.block_id_counter += 1

        if not self.free_list:
            return None

        # Start from roving pointer or beginning if None
        start_node = self.next_fit_ptr if self.next_fit_ptr else self.free_list
        current = start_node
        first_pass = True

        while current:
            if current.size >= size:
                # Found a fitting block
                allocated_start = current.start

                # Update free list
                if current.size == size:  # Exact fit
                    # Remove current node from free list
                    # Find previous node
                    prev = None
                    temp = self.free_list
                    while temp and temp != current:
                        prev = temp
                        temp = temp.next

                    if prev:
                        prev.next = current.next
                    else:
                        self.free_list = current.next

                    # Update next_fit_ptr to next node or beginning
                    self.next_fit_ptr = current.next if current.next else self.free
_list
                else:  # Split the block
                    current.start += size
                    current.size -= size
                    # Update roving pointer to current block (the remainder)
                    self.next_fit_ptr = current

                # Record allocation
                self.allocated_blocks[block_id] = (allocated_start, size)
```

```
                    self.allocation_sequence.append(('allocate', block_id, size, 'next_
fit'))
                    return allocated_start

                # Move to next block
                current = current.next

                # If we've reached the end, wrap around to beginning
                if not current and first_pass:
                    current = self.free_list
                    first_pass = False

                # If we've come back to start_node, break to avoid infinite loop
                if current == start_node and not first_pass:
                    break

            return None  # No block found

        def free_block(self, block_id):
            """Free an allocated block and merge with adjacent free blocks"""
            if block_id not in self.allocated_blocks:
                return False

            start, size = self.allocated_blocks[block_id]
            del self.allocated_blocks[block_id]

            # Create new free block
            new_block = FreeBlock(start, size)

            # Insert into free list in sorted position (by start address)
            if not self.free_list or start < self.free_list.start:
                # Insert at beginning
                new_block.next = self.free_list
                self.free_list = new_block
            else:
                # Find insertion point
                current = self.free_list
                prev = None
                while current and current.start < start:
                    prev = current
                    current = current.next

                # Insert between prev and current
                new_block.next = current
                if prev:
                    prev.next = new_block

            # Coalesce adjacent blocks
            self.coalesce_free_blocks()

            # Update next_fit_ptr if it points to a block that was freed
            # If next_fit_ptr is None or points to a block that's no longer in free lis
t,
            # reset it to free_list
            if self.next_fit_ptr:
```

```
            # Check if next_fit_ptr is still in the free list
            temp = self.free_list
            found = False
            while temp:
                if temp == self.next_fit_ptr:
                    found = True
                    break
                temp = temp.next
            if not found:
                self.next_fit_ptr = self.free_list
        else:
            self.next_fit_ptr = self.free_list

        self.allocation_sequence.append(('free', block_id, size, None))
        return True


# ==================== EXPERIMENT 1: ALLOCATION TRACE ====================
def experiment1_allocation_trace():
    """Run Experiment 1: Fixed allocation/free sequence"""
    print("\n" + "=" * 70)
    print("EXPERIMENT 1: Allocation Trace")
    print("=" * 70)
    print("Sequence: [10, 5, 20, -5, 12, -10, 8, 6, 7, 3, 10]")
    print("Positive = allocate, Negative = free that size\n")

    sequence = [10, 5, 20, -5, 12, -10, 8, 6, 7, 3, 10]

    # Create three allocators, one for each algorithm
    allocators = {
        'Best Fit': MemoryAllocator(100),
        'Worst Fit': MemoryAllocator(100),
        'Next Fit': MemoryAllocator(100)
    }

    # Track allocations for each algorithm
    allocations_by_size = {
        'Best Fit': {},
        'Worst Fit': {},
        'Next Fit': {}
    }

    for step, request in enumerate(sequence):
        print(f"\n{'=' * 50}")
        print(f"Step {step + 1}: Request {request}")
        print('=' * 50)

        for algo_name, allocator in allocators.items():
            if request > 0:  # Allocation
                # Generate block ID
                block_id = allocator.block_id_counter

                # Store mapping for later freeing
                if request not in allocations_by_size[algo_name]:
                    allocations_by_size[algo_name][request] = []
```

```
                    allocations_by_size[algo_name][request].append(block_id)

                    # Perform allocation based on algorithm
                    if algo_name == 'Best Fit':
                        result = allocator.allocate_best_fit(request, block_id)
                    elif algo_name == 'Worst Fit':
                        result = allocator.allocate_worst_fit(request, block_id)
                    else:  # Next Fit
                        result = allocator.allocate_next_fit(request, block_id)

                    allocator.print_free_list(algo_name)

                else:  # Freeing (negative request)
                    size_to_free = -request

                    if (size_to_free in allocations_by_size[algo_name] and
                            allocations_by_size[algo_name][size_to_free]):
                        # Free the first block of this size
                        block_id = allocations_by_size[algo_name][size_to_free].pop(0)
                        allocator.free_block(block_id)
                        allocator.print_free_list(algo_name)
                    else:
                        # No block of this size to free
                        allocator.print_free_list(algo_name)

    print("\n" + "=" * 70)
    print("EXPERIMENT 1 ANALYSIS")
    print("=" * 70)

    print("\n1. BEHAVIORAL DIFFERENCES:")
    print("   • BEST FIT:")
    print("     - Tends to allocate from smallest fitting blocks")
    print("     - Creates many small fragments quickly")
    print("     - External fragmentation appears early")

    print("\n   • WORST FIT:")
    print("     - Always picks the largest available block")
    print("     - Leaves larger leftover blocks")
    print("     - Better at preserving large contiguous spaces")

    print("\n   • NEXT FIT:")
    print("     - Starts search from last allocation position")
    print("     - Distributes allocations around memory")
    print("     - More uniform fragmentation pattern")

    print("\n2. FREE LIST DIVERGENCE:")
    print("   • As fragmentation grows, algorithms produce")
    print("     completely different free list structures")
    print("   • Best Fit: Many small scattered blocks")
    print("   • Worst Fit: Fewer but larger blocks")
    print("   • Next Fit: More evenly distributed blocks")


# ==================== EXPERIMENT 2: FRAGMENTATION TEST ====================
def experiment2_fragmentation_test():
```

```python
    """Run Experiment 2: Random allocations then large allocation"""
    print("\n" + "=" * 70)
    print("EXPERIMENT 2: Fragmentation Test")
    print("=" * 70)
    print("\nSteps:")
    print("1. Perform 12 random allocations (sizes 3-12)")
    print("2. Free exactly 4 previously allocated blocks at random")
    print("3. Attempt one large allocation of size 25")
    print("\n" + "-" * 70)

    # Set random seed for reproducible results
    random.seed(42)

    algorithms = ['Best Fit', 'Worst Fit', 'Next Fit']
    results = {}

    for algo_name in algorithms:
        print(f"\n{'=' * 50}")
        print(f"ALGORITHM: {algo_name}")
        print('=' * 50)

        # Create allocator
        allocator = MemoryAllocator(100)

        # Step 1: 12 random allocations
        print("\n1. Making 12 random allocations (size 3-12):")
        allocated_blocks = []  # List of (block_id, size)

        for i in range(12):
            size = random.randint(3, 12)
            block_id = allocator.block_id_counter

            if algo_name == 'Best Fit':
                result = allocator.allocate_best_fit(size, block_id)
            elif algo_name == 'Worst Fit':
                result = allocator.allocate_worst_fit(size, block_id)
            else:  # Next Fit
                result = allocator.allocate_next_fit(size, block_id)

            if result is not None:
                allocated_blocks.append((block_id, size))
                # print(f"   Allocated {size} units at address {result}")

        allocator.print_free_list("After 12 allocs")

        # Step 2: Free 4 random blocks
        print("\n2. Freeing 4 random blocks:")
        if len(allocated_blocks) >= 4:
            to_free = random.sample(allocated_blocks, 4)
            freed_info = []
            for block_id, size in to_free:
                allocator.free_block(block_id)
                freed_info.append(f"size {size}")
                # Remove from allocated_blocks
                allocated_blocks = [b for b in allocated_blocks if b[0] != block_i
```

```
d]
            print(f"   Freed blocks: {', '.join(freed_info)}")

        allocator.print_free_list("After freeing 4")

        # Step 3: Attempt to allocate 25 units
        print("\n3. Attempting to allocate size 25:")
        if algo_name == 'Best Fit':
            result = allocator.allocate_best_fit(25)
        elif algo_name == 'Worst Fit':
            result = allocator.allocate_worst_fit(25)
        else:  # Next Fit
            result = allocator.allocate_next_fit(25)

        # Analyze free list
        current = allocator.free_list
        total_free = 0
        largest_block = 0
        block_count = 0

        while current:
            total_free += current.size
            if current.size > largest_block:
                largest_block = current.size
            block_count += 1
            current = current.next

        if result is not None:
            print(f"   ✅ SUCCESS: Allocated at address {result}")
            success = True
            # Free it for consistency
            allocator.free_block(allocator.block_id_counter - 1)
        else:
            print(f"   ❌ FAILED: Cannot allocate 25 units")
            success = False

        print(f"\n   Free List Analysis:")
        print(f"   • {block_count} free blocks")
        print(f"   • Total free memory: {total_free} units")
        print(f"   • Largest contiguous block: {largest_block} units")

        if success:
            print(f"   • Reason: Found block ≥ 25 units ({largest_block})")
        else:
            print(f"   • Reason: External fragmentation")
            print(f"     (Total free: {total_free}, but no block ≥ 25)")

        results[algo_name] = {
            'success': success,
            'total_free': total_free,
            'largest_block': largest_block,
            'block_count': block_count
        }

    print("\n" + "=" * 70)
```

```
        print("EXPERIMENT 2 SUMMARY")
        print("=" * 70)

        print("\nRESULTS:")
        for algo_name, result in results.items():
            status = "SUCCESS" if result['success'] else "FAILED"
            print(f"  {algo_name:10s}: {status}")
            print(f"    Free blocks: {result['block_count']}, Largest: {result['largest
_block']}")

        print("\nKEY INSIGHTS:")
        print("1. BEST FIT often fails due to external fragmentation")
        print("   • Creates many small blocks")
        print("   • Even with enough total free memory, no single block is large enoug
h")

        print("\n2. WORST FIT usually succeeds")
        print("   • Preserves larger free blocks")
        print("   • Better chance of having a block ≥ 25 units")

        print("\n3. NEXT FIT results vary")
        print("   • Depends on roving pointer position")
        print("   • May succeed or fail based on fragmentation distribution")


    # ==================== EXPERIMENT 3: SPEED TEST ====================
    def experiment3_speed_test():
        """Run Experiment 3: Performance comparison"""
        print("\n" + "=" * 70)
        print("EXPERIMENT 3: Speed Test")
        print("=" * 70)
        print("\nProcedure:")
        print("1. Repeat 200 times:")
        print("   a. Allocate random block (size 1-10)")
        print("   b. Free one previously allocated block")
        print("2. Measure total execution time")
        print("\n" + "-" * 70)

        random.seed(123)  # For reproducible results
        algorithms = ['Best Fit', 'Worst Fit', 'Next Fit']
        results = {}

        for algo_name in algorithms:
            print(f"\nTesting {algo_name}...")

            allocator = MemoryAllocator(100)
            allocated_blocks = []  # Track block IDs for freeing
            allocations_made = 0
            frees_made = 0

            # Start timing
            start_time = time.perf_counter()

            for i in range(200):
                # Allocate random block
```

```
                size = random.randint(1, 10)
                block_id = allocator.block_id_counter

                if algo_name == 'Best Fit':
                    result = allocator.allocate_best_fit(size, block_id)
                elif algo_name == 'Worst Fit':
                    result = allocator.allocate_worst_fit(size, block_id)
                else:  # Next Fit
                    result = allocator.allocate_next_fit(size, block_id)

                if result is not None:
                    allocated_blocks.append(block_id)
                    allocations_made += 1

                # Free a block if we have any allocated (with 50% probability)
                if allocated_blocks and random.random() < 0.5:
                    block_to_free = random.choice(allocated_blocks)
                    allocator.free_block(block_to_free)
                    allocated_blocks.remove(block_to_free)
                    frees_made += 1

            # End timing
            end_time = time.perf_counter()
            elapsed = end_time - start_time

            results[algo_name] = {
                'time': elapsed,
                'allocations': allocations_made,
                'frees': frees_made
            }

            print(f"  Time: {elapsed:.6f} seconds")
            print(f"  Allocations: {allocations_made}, Frees: {frees_made}")
            print(f"  Operations/sec: {(allocations_made + frees_made) / elapsed:.0f}")

        print("\n" + "=" * 70)
        print("SPEED TEST RESULTS")
        print("=" * 70)

        # Find fastest and slowest
        fastest_algo = min(results, key=lambda x: results[x]['time'])
        slowest_algo = max(results, key=lambda x: results[x]['time'])

        print("\nRanking (fastest to slowest):")
        for algo_name in sorted(results, key=lambda x: results[x]['time']):
            time_val = results[algo_name]['time']
            ops_sec = (results[algo_name]['allocations'] + results[algo_name]['frees'])
/ time_val
            marker = ""
            if algo_name == fastest_algo:
                marker = " (FASTEST)"
            elif algo_name == slowest_algo:
                marker = " (SLOWEST)"
            print(f"  {algo_name:10s}: {time_val:.6f}s ({ops_sec:.0f} ops/sec){marke
r}")
```

```
    print("\n" + "=" * 70)
    print("PERFORMANCE ANALYSIS")
    print("=" * 70)

    print("\n1. WHY NEXT FIT IS FASTEST:")
    print("   • Search Strategy: Starts from last position, not beginning")
    print("   • Average Search Length: ~50% of free list")
    print("   • Time Complexity: O(n/2) on average")

    print("\n2. WHY BEST FIT IS SLOWEST:")
    print("   • Search Strategy: Must scan ENTIRE free list every time")
    print("   • Comparison: Compares all blocks to find smallest fit")
    print("   • Time Complexity: O(n) always")

    print("\n3. WORST FIT PERFORMANCE:")
    print("   • Similar to Best Fit: Scans entire list")
    print("   • Slightly faster: Can stop early in some implementations")
    print("   • Still O(n) complexity")

    print("\nSCANNING ANALYSIS PER ALLOCATION:")
    print("  Algorithm    | Avg. Nodes Scanned | Complexity")
    print("  ------------ | ------------------ | ----------")
    print("  Next Fit     | ~n/2               | O(n/2)")
    print("  Best Fit     | n                  | O(n)")
    print("  Worst Fit    | n                  | O(n)")

    print("\nCONCLUSION:")
    print("• For speed: Next Fit is the clear winner")
    print("• For fragmentation: Worst Fit performs better")
    print("• For space utilization: Best Fit is most efficient")
    print("• Real systems often use hybrid approaches")


# ==================== MAIN PROGRAM ====================
def main():
    """Main program with menu interface"""
    print("=" * 80)
    print("MEMORY ALLOCATION ALGORITHMS - COMPLETE IMPLEMENTATION")
    print("Question 3: Best Fit, Worst Fit, and Next Fit Comparison")
    print("=" * 80)

    while True:
        print("\n" + "=" * 80)
        print("MAIN MENU")
        print("=" * 80)
        print("\nSelect an option:")
        print("1. Run Experiment 1: Allocation Trace")
        print("2. Run Experiment 2: Fragmentation Test")
        print("3. Run Experiment 3: Speed Test")
        print("4. Run ALL Experiments")
        print("5. Exit Program")
        print("\n" + "-" * 80)

        choice = input("Enter your choice (1-5): ").strip()
```

```
649
650         if choice == "1":
651             experiment1_allocation_trace()
652             input("\nPress Enter to continue...")
653
654         elif choice == "2":
655             experiment2_fragmentation_test()
656             input("\nPress Enter to continue...")
657
658         elif choice == "3":
659             experiment3_speed_test()
660             input("\nPress Enter to continue...")
661
662         elif choice == "4":
663             print("\nRunning all experiments sequentially...\n")
664             experiment1_allocation_trace()
665             print("\n" + "=" * 80)
666             experiment2_fragmentation_test()
667             print("\n" + "=" * 80)
668             experiment3_speed_test()
669             print("\n" + "=" * 80)
670             print("ALL EXPERIMENTS COMPLETED!")
671             input("\nPress Enter to return to menu...")
672
673         elif choice == "5":
674             print("\nThank you for using the Memory Allocator Simulator!")
675             print("Goodbye!")
676             break
677
678         else:
679             print("Invalid choice. Please enter 1-5.")
680
681
682 # ==================== PROGRAM ENTRY POINT ====================
683 if __name__ == "__main__":
684     # Run the main program
685     main()
```

C. Experiment 1 – Allocation Trace (5 points)

Using the fixed sequence:

$$[10, 5, 20, -5, 12, -10, 8, 6, 7, 3, 10]$$

Positive numbers represent allocations of that size. Negative numbers represent freeing the block that was allocated with that size.

Example: A request of -5 frees the *first* block of size 5 that was previously allocated.

This sequence intentionally creates early fragmentation so that **Best Fit**, **Worst Fit**, and **Next Fit** will produce clearly different free-list shapes. For each request (allocation or free), print the **entire free list** for each algorithm.

```
============================================================
EXPERIMENT 1: Allocation Trace
============================================================
Sequence: [10, 5, 20, -5, 12, -10, 8, 6, 7, 3, 10]
Positive = allocate, Negative = free that size



================================================
Step 1: Request 10
================================================
Best Fit   Free List: [ 10- 99]( 90)
Worst Fit  Free List: [ 10- 99]( 90)
Next Fit   Free List: [ 10- 99]( 90)

================================================
Step 2: Request 5
================================================
Best Fit   Free List: [ 15- 99]( 85)
Worst Fit  Free List: [ 15- 99]( 85)
Next Fit   Free List: [ 15- 99]( 85)
```

```
================================================
Step 3: Request 20
================================================
Best Fit   Free List: [ 35- 99]( 65)
Worst Fit  Free List: [ 35- 99]( 65)
Next Fit   Free List: [ 35- 99]( 65)

================================================
Step 4: Request -5
================================================
Best Fit   Free List: [ 15- 99]( 85)
Worst Fit  Free List: [ 15- 99]( 85)
Next Fit   Free List: [ 15- 99]( 85)
```

```
================================================
Step 5: Request 12
================================================
Best Fit   Free List: [ 27- 99]( 73)
Worst Fit  Free List: [ 27- 99]( 73)
Next Fit   Free List: [ 15- 99]( 85)

================================================
Step 6: Request -10
================================================
Best Fit   Free List: [ 15- 99]( 85)
Worst Fit  Free List: [ 15- 99]( 85)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)

================================================
Step 7: Request 8
================================================
Best Fit   Free List: [ 23- 99]( 77)
Worst Fit  Free List: [ 23- 99]( 77)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)

================================================
Step 8: Request 6
================================================
Best Fit   Free List: [ 29- 99]( 71)
Worst Fit  Free List: [ 29- 99]( 71)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)
```

```
================================================
Step 9: Request 7
================================================
Best Fit   Free List: [ 36- 99]( 64)
Worst Fit  Free List: [ 36- 99]( 64)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)

================================================
Step 10: Request 3
================================================
Best Fit   Free List: [ 39- 99]( 61)
Worst Fit  Free List: [ 39- 99]( 61)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)

================================================
Step 11: Request 10
================================================
Best Fit   Free List: [ 49- 99]( 51)
Worst Fit  Free List: [ 49- 99]( 51)
Next Fit   Free List: [ 15- 99]( 85) → [ 35- 46]( 12)
```

```
================================================================================
EXPERIMENT 1 ANALYSIS
================================================================================

1. BEHAVIORAL DIFFERENCES:
   • BEST FIT:
     - Tends to allocate from smallest fitting blocks
     - Creates many small fragments quickly
     - External fragmentation appears early

   • WORST FIT:
     - Always picks the largest available block
     - Leaves larger leftover blocks
     - Better at preserving large contiguous spaces

   • NEXT FIT:
     - Starts search from last allocation position
     - Distributes allocations around memory
     - More uniform fragmentation pattern

2. FREE LIST DIVERGENCE:
   • As fragmentation grows, algorithms produce
     completely different free list structures
   • Best Fit: Many small scattered blocks
   • Worst Fit: Fewer but larger blocks
   • Next Fit: More evenly distributed blocks
```

Answer the following:
Describe at least *one clear behavioral difference* among Best Fit, Worst Fit, and Next Fit —for example, in terms of

- which part of memory they tend to allocate from,
- how quickly each algorithm creates small fragments,
- whether they cluster allocations or spread them out,
- how their free lists diverge as fragmentation grows.

Feature	Best fit	Worse fit	Next fit
Primary Goal	Minimize leftover space	Maximize leftover space	Increase search speed
Search Range	Entire list	Entire list	from last position
Leftover Result	Tiny slivers	Large useful blocks	Distributed fragments
Complexity	O(n)	O(n)	$O(\frac{n}{2})$

D. Experiment 2 – Fragmentation Test (5 points)

For each algorithm:

1. perform 12 random allocations (sizes 3–12),
2. free exactly 4 previously allocated blocks at random,
3. attempt one large allocation of size 25.

```
======================================================================
EXPERIMENT 2: Fragmentation Test
======================================================================

Steps:
1. Perform 12 random allocations (sizes 3-12)
2. Free exactly 4 previously allocated blocks at random
3. Attempt one large allocation of size 25

----------------------------------------------------------------------


================================================
ALGORITHM: Best Fit
================================================

1. Making 12 random allocations (size 3-12):
After 12 allocs Free List: [ 74- 99]( 26)

2. Freeing 4 random blocks:
   Freed blocks: size 4, size 3, size 6, size 12
After freeing 4 Free List: [ 71- 99]( 29)

3. Attempting to allocate size 25:
   ✅ SUCCESS: Allocated at address 71

   Free List Analysis:
   • 1 free blocks
   • Total free memory: 4 units
   • Largest contiguous block: 4 units
   • Reason: Found block ≥ 25 units (4)
================================================
ALGORITHM: Worst Fit
================================================

1. Making 12 random allocations (size 3-12):
After 12 allocs Free List: [ 98- 99](  2)

2. Freeing 4 random blocks:
   Freed blocks: size 3, size 9, size 11, size 6
After freeing 4 Free List: [ 91- 99](  9)

3. Attempting to allocate size 25:
   ❌ FAILED: Cannot allocate 25 units

   Free List Analysis:
   • 1 free blocks
   • Total free memory: 9 units
   • Largest contiguous block: 9 units
   • Reason: External fragmentation
     (Total free: 9, but no block ≥ 25)
```

```
================================================
ALGORITHM: Next Fit
================================================

1. Making 12 random allocations (size 3-12):
After 12 allocs Free List: [ 78- 99]( 22)

2. Freeing 4 random blocks:
   Freed blocks: size 3, size 8, size 8, size 6
After freeing 4 Free List: [ 75- 99]( 25)

3. Attempting to allocate size 25:
   ✅ SUCCESS: Allocated at address 75

   Free List Analysis:
   • 0 free blocks
   • Total free memory: 0 units
   • Largest contiguous block: 0 units
   • Reason: Found block ≥ 25 units (0)
======================================================================
EXPERIMENT 2 SUMMARY
======================================================================

RESULTS:
  Best Fit  : SUCCESS
    Free blocks: 1, Largest: 4
  Worst Fit : FAILED
    Free blocks: 1, Largest: 9
  Next Fit  : SUCCESS
    Free blocks: 0, Largest: 0
```

After these operations, print the final free list. Report whether each algorithm succeeds or fails at the large allocation, and explain why in terms of:

- external fragmentation,
- the algorithm's search strategy (first few blocks, largest block, next pointer, etc.),
- how earlier choices increased or reduced the chance of finding a 25-byte segment.

Best-fit approach often fails because of the external fragmentation. It creates lots of small blocks. Even totally the required free space exist no single blocks large enough.

Worse-fit usually succeeds it preserves longer free blocks. But it depends if there is a block has more than 25 units space or not.

Next-fit results vary because of the moving pointer position. May succeed or not based on fragmentation distribution.

E. Experiment 3 – Speed Test (5 points)

Repeat the following 200 times:

1. allocate a random block (size 1–10),
2. free one previously allocated block.

Measure the total time for Best Fit, Worst Fit, and Next Fit.

```
======================================================================
EXPERIMENT 3: Speed Test
======================================================================

Procedure:
1. Repeat 200 times:
   a. Allocate random block (size 1-10)
   b. Free one previously allocated block
2. Measure total execution time

----------------------------------------------------------------------

Testing Best Fit...
  Time: 0.000503 seconds
  Allocations: 90, Frees: 90
  Operations/sec: 357739

Testing Worst Fit...
  Time: 0.000485 seconds
  Allocations: 68, Frees: 68
  Operations/sec: 280650

Testing Next Fit...
  Time: 0.000321 seconds
  Allocations: 53, Frees: 53
  Operations/sec: 330259
```

Answer the following:

Which algorithm is fastest and which is slowest? Explain your conclusion based on how much of the linked list each algorithm scans.

Next-fit fastest → starts from last position, not beginning
Best-fit slowest → scans entire free list every time
Worst-fit second → slightly faster than best-fit
because it can stop early in some implementation

Question 4 – GitHub Repository (10 points)

A. Repository Link (5 points)

For this question, you must submit your implementation using a public **GitHub repository**. Your repository should contain:

- All source code for the three allocation algorithms (Best Fit, Worst Fit, Next Fit),
- A clear directory structure,
- A `README.md` file explaining how to run your code,
- Any scripts you used for running experiments or generating results.

At the end of your submission, include a direct link to your GitHub repository.

https://github.com/Raphael-is-Coding

B. Reproducible GitHub Repositories (5 points)

In this part, briefly explain the importance of having a **reproducible GitHub repository** when submitting programming work. Your explanation should address:

- **Why reproducibility matters** (e.g., allowing others to run your code, verify results, or build on your work).
- **How you would make your repository reproducible** even with limited time. For example, you may mention simple steps such as:
  - including a short `README.md` with run instructions,
  - listing required packages or versions,
  - keeping files organized so the code can be executed easily.

You do **not** need to fully implement reproducibility features — just describe the key ideas and your approach.

I keep my Github repository clean, readable easily understandable because;

Debugging and accountability perspective if someone make mistake on a project someone else could see the problem and give brief on that.

Reusability perspective → we can last of the project codes and ~~and~~ ~~out~~ unintentionally delete files so its better to have it on Github platform

Updates → You can update the code later and continue where you were.