Overview

My goals of the assignment is to construct different disk algorithms, and compare their performance. The disk algorithms are FCFS, SSTF, SCAN, C-SCAN, LOOK, CLOOK and Optimal. Then I will describe them in details.

What does each algorithm do? Describe its behavior.

1. FCFS (First-Come, First-Served)
    - Behavior: FIFO simply queues processes in the order that they arrive in the ready queue.
    - Execution: It moves the disk head to each request in the sequence they are received without reordering.

```
void FCFS(int* requests, int head, int *total_movement, double *average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    for(int i = 0; i < REQUESTS; i++){
        *total_movement += abs(requests[i] - head);
        head = requests[i];
    }

    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

2. SSTF
    - Behavior: Selects the request closest to the current head position.
    - Execution: It reduces the total seek time by always choosing the nearest request.
    - In this assignment, we assume that the head is 0, which means that SSTF will perform like OPTIMAL.
    - The implementation of SSTF is like greedy. It always choose the current best solution.

```c
void SSTF(int* requests, int head, int *total_movement, double *average_latency)
{
    int visited[REQUESTS] = {0};    // Record the requests which have been visited
    *total_movement = 0;
    *average_latency = 0;

    for(int i = 0; i < REQUESTS; i++){
        int min_distance = CYLINDERS;
        int index = -1;

        for(int j = 0; j < REQUESTS; j++){
            if(!visited[j] && abs(requests[j] - head) < min_distance){
                min_distance = abs(requests[j] - head);
                index = j;    // Record the next selected request
            }
        }

        if (index != -1) {
            visited[index] = 1;    // Declare that the request has been serviced
            *total_movement += abs(requests[index] - head);
            head = requests[index]; // Update head position
        }
    }

    *average_latency = (*total_movement / 100.0) * 1.0; // Calculate average latency
}
```

3. SCAN
    - Behavior: Moves the disk head in one direction, fulfilling all requests until it reaches the end, then reverses direction.
    - Execution: It scans back and forth across the disk, servicing requests in the order they are encountered.
    - In this assignment we assume that the head is 0, which means that the direction is always right; however, in my implementation, I still check which direction is first.
    - The direction is determined by the position between the head and the first request. If the position of head is smaller than the first request, then the direction will be right; otherwise it will be left.

```c
void SCAN(int* requests, int head, int *total_movement, double *average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    // Split the requests into left array and right array
    int left[REQUESTS + 1], right[REQUESTS + 1];
    int left_size = 0, right_size = 0;

    // Determine the initial direction
    char direction[6];
    if (head < requests[0]) {
        strcpy(direction, "right");
    }
    else {
        strcpy(direction, "left");
    }

    // Appending end values which has to be visited before reversing the direction
    if (strcmp(direction, "left") == 0) {
        left[left_size++] = 0;
    }
    else if (strcmp(direction, "right") == 0) {
        right[right_size++] = CYLINDERS - 1;
    }
```

```
    for (int i = 0; i < REQUESTS; i++) {
        if (requests[i] < head) {
            left[left_size++] = requests[i];
        }

        if (requests[i] > head) {
            right[right_size++] = requests[i];
        }
    }

    // Sort the left array and the right array
    qsort(left, left_size, sizeof(int), cmp);
    qsort(right, right_size, sizeof(int), cmp);

    int cur_track;  // record the request which is serviced currently
```

```
    // Run the while loop two times, one by one scanning right and left of the head
    int run = 2;
    while (run--) {
        if (strcmp(direction, "left") == 0) {
            for (int i = left_size - 1; i >= 0; i--) {
                cur_track = left[i];

                // Increase the total count
                *total_movement += abs(cur_track - head);

                // Accessed track is now the new head
                head = cur_track;
            }
            // set the direction to the right
            strcpy(direction, "right");
        }
```

```
        else if (strcmp(direction, "right") == 0) {
            for (int i = 0; i < right_size; i++) {
                cur_track = right[i];

                // Increase the total count
                *total_movement += abs(cur_track - head);

                // Accessed track is now new head
                head = cur_track;
            }
            // set the direction to the left
            strcpy(direction, "left");
        }
    }

    // Calculate average latency
    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

4. C-SCAN
    - Behavior: Similar to SCAN, but after reaching the end, it jumps back to the beginning without servicing requests on the return trip.
    - Execution: This provides a more uniform wait time compared to SCAN by treating the disk as a circular list.

```c
void CSCAN(int* requests, int head, int *total_movement, double *average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    int left[REQUESTS + 1], right[REQUESTS + 1];
    int left_size = 0, right_size = 0;

    // Appending end values which has to be visited before reversing the direction
    left[left_size++] = 0;
    right[right_size++] = CYLINDERS - 1;

    // Tracks on the left of the head will be serviced once the head comes back to the beginning.
    for (int i = 0; i < REQUESTS; i++) {
        if (requests[i] < head) {
            left[left_size++] = requests[i];
        }
        if (requests[i] > head) {
            right[right_size++] = requests[i];
        }
    }

    // Sorting left and right arrays
    qsort(left, left_size, sizeof(int), cmp);
    qsort(right, right_size, sizeof(int), cmp);

    int cur_track;  // record the request which is serviced currently
```

```c
    // First service the requests on the right side of the head
    for (int i = 0; i < right_size; i++) {
        cur_track = right[i];
        // Increase the total count
        *total_movement += abs(cur_track - head);
        // Update the head
        head = cur_track;
    }
    // Once reached the right end, jump to the beginning
    head = 0;
    // Jumps to the head and adding the distance from right end to left end (+CYLINDERS - 1)
    *total_movement += (CYLINDERS - 1);
    // Now service the requests which are in left array
    for (int i = 0; i < left_size; i++) {
        cur_track = left[i];
        // Increase the total count
        *total_movement += abs(cur_track - head);
        // Update the head
        head = cur_track;
    }
    // Calculate average latency
    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

5. LOOK
    - Behavior: Similar to SCAN but stops moving in one direction once there are no more requests in that direction.
    - Execution: Moves the head in one direction fulfilling all requests until there are no more requests in that direction, then reverses.

```c
void LOOK(int* requests, int head, int* total_movement, double* average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    int left[REQUESTS], right[REQUESTS];
    int left_size = 0, right_size = 0;

    // Determine the initial direction
    char direction[6];
    if (head < requests[0]) {
        strcpy(direction, "right");
    }
    else {
        strcpy(direction, "left");
    }

    // Appending values which are currently at left and right direction from the head
    for (int i = 0; i < REQUESTS; i++) {
        if (requests[i] < head) {
            left[left_size++] = requests[i];
        }
        if (requests[i] > head) {
            right[right_size++] = requests[i];
        }
    }
```

```c
    // Sorting left and right arrays
    qsort(left, left_size, sizeof(int), cmp);
    qsort(right, right_size, sizeof(int), cmp);

    int cur_track;  // record the request which is serviced currently

    // Run the while loop two times, one by one scanning right and left side of the head
    int run = 2;
    while (run--) {
        if (strcmp(direction, "left") == 0) {
            for (int i = left_size - 1; i >= 0; i--) {
                cur_track = left[i];

                // Increase the total count
                *total_movement += abs(cur_track - head);

                // Update the head
                head = cur_track;
            }
            // Reversing the direction
            strcpy(direction, "right");
        }
```

```c
        else if (strcmp(direction, "right") == 0) {
            for (int i = 0; i < right_size; i++) {
                cur_track = right[i];

                // Increase the total count
                *total_movement += abs(cur_track - head);

                // Update the head
                head = cur_track;
            }
            // Reversing the direction
            strcpy(direction, "left");
        }
    }

    // Calculate average latency
    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

6. CLOOK

- Behavior: Similar to CSCAN but stops once there are no more requests in the current direction and jumps back to the first request in the other direction.
- Execution: Services all requests in one direction then jumps back to the first request in the other direction.

```c
void CLOOK(int* requests, int head, int* total_movement, double* average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    int left[REQUESTS], right[REQUESTS];
    int left_size = 0, right_size = 0;

    // Split the requests into left array and right array
    for (int i = 0; i < REQUESTS; i++) {
        if (requests[i] < head) {
            left[left_size++] = requests[i];
        }
        else if (requests[i] > head) {
            right[right_size++] = requests[i];
        }
    }

    // Sort the left array and the right array
    qsort(left, left_size, sizeof(int), cmp);
    qsort(right, right_size, sizeof(int), cmp);

    int cur_track;  // record the request which is serviced currently
```

```c
    // First deal with the right array
    for (int i = 0; i < right_size; i++) {
        cur_track = right[i];

        *total_movement += abs(cur_track - head);

        // Update the head
        head = cur_track;
    }

    // Once reach the end of the right array, jump to the first one of the left array
    if (left_size > 0) {
        *total_movement += abs(head - left[0]);
        head = left[0];
    }

    // Deal with the left requests
    for (int i = 0; i < left_size; i++) {
        cur_track = left[i];

        *total_movement += abs(cur_track - head);

        // Update the head
        head = cur_track;
    }

    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

7. OPTIMAL
   - Behavior: Processes requests in an optimal sequence that minimizes the total head movement.
   - Execution: It sorts the requests and processes them in sorted order, which is theoretically the best for minimizing movement.
   - First I use ***qsort*** to sort the requests, and then process them.
   - ***qsort*** is a built-in function in C, and it will sort the requests by self-

defined function ***cmp()***.

```c
void OPTIMAL(int* requests, int head, int* total_movement, double* average_latency)
{
    *total_movement = 0;
    *average_latency = 0;

    int sorted_requests[REQUESTS];
    for(int i = 0; i < REQUESTS; i++) {
        sorted_requests[i] = requests[i];
    }

    // Sort the requests, and put the result in the sorted_requests
    qsort(sorted_requests, REQUESTS, sizeof(int), cmp);

    for(int i = 0; i < REQUESTS; i++) {
        *total_movement += abs(sorted_requests[i] - head);
        head = sorted_requests[i];
    }

    *average_latency = (*total_movement / 100.0) * 1.0;
}
```

What is the objective / goal of each algorithm?

1. FCFS: Fairness and simplicity. It aims to serve requests in the order they arrive.
2. SSTF: Minimizes seek time by always selecting the nearest request.
3. SCAN: Provides a more equitable service time by scanning back and forth across the disk.
4. CSCAN: Provides a more uniform wait time than SCAN by treating the disk as a circular list.
5. LOOK: Reduces unnecessary movement by stopping once there are no more requests in the current direction.
6. CLOOK: Similar to LOOK, but it also ensures more uniform wait times by jumping back to the first request after reaching the end.
7. OPTIMAL: Minimizes the total head movement by processing requests in an optimal order.

What are the strengths and weaknesses of each disk-scheduling algorithm?

1. FCFS
   - Strengths: Simple and fair.
   - Weaknesses: Can result in high seek time if requests are far apart

(convoy effect).

2. SSTF
   - Strengths: Reduces overall seek time.
   - Weaknesses: Can cause starvation of requests that are far from the current head position.
3. SCAN
   - Strengths: Reduces variance in wait time compared to FCFS and SSTF.
   - Weaknesses: Can cause longer wait times for requests just missed when the head reverses direction.
4. CSCAN
   - Strengths: Provides more uniform wait time than SCAN.
   - Weaknesses: Can be less efficient than SCAN due to the jump back to the beginning.
5. LOOK
   - Strengths: Reduces unnecessary head movement compared to SCAN.
   - Weaknesses: Similar to SCAN, can cause longer wait times for requests just missed.
6. CLOOK
   - Strengths: Provides uniform wait times and reduces unnecessary movement.
   - Weaknesses: Similar to CSCAN, but with the same issue of jumping back to the beginning.
7. OPTIMAL
   - Strengths: Minimizes total head movement.
   - Weaknesses: Requires knowledge of all requests in advance, which is not always practical.

What is the average latency of each algorithm (assume 1ms for every 100 cylinders)?

The average latency can be calculated from the total head movement divided by the number of requests, and then converted to milliseconds.

We first assume that the position of head is at 0.

```
C:\Users\315>gcc -o test test.c

C:\Users\315>test 0
FCFS: Total Head Movement = 1595646, Average Latency = 15956.46 ms
SSTF: Total Head Movement = 4994, Average Latency = 49.94 ms
SCAN: Total Head Movement = 4999, Average Latency = 49.99 ms
CSCAN: Total Head Movement = 9998, Average Latency = 99.98 ms
LOOK: Total Head Movement = 4994, Average Latency = 49.94 ms
CLOOK: Total Head Movement = 4994, Average Latency = 49.94 ms
OPTIMAL: Total Head Movement = 4994, Average Latency = 49.94 ms
```

We can see that FCFS has the largest head movement and average latency. It is simple to implement, but it is not a good disk algorithm.

The other disk algorithm has the same head movement and average latency, that is because we assume that the head is always beginning at 0. If we alternate the position of head, we will get the different answer.

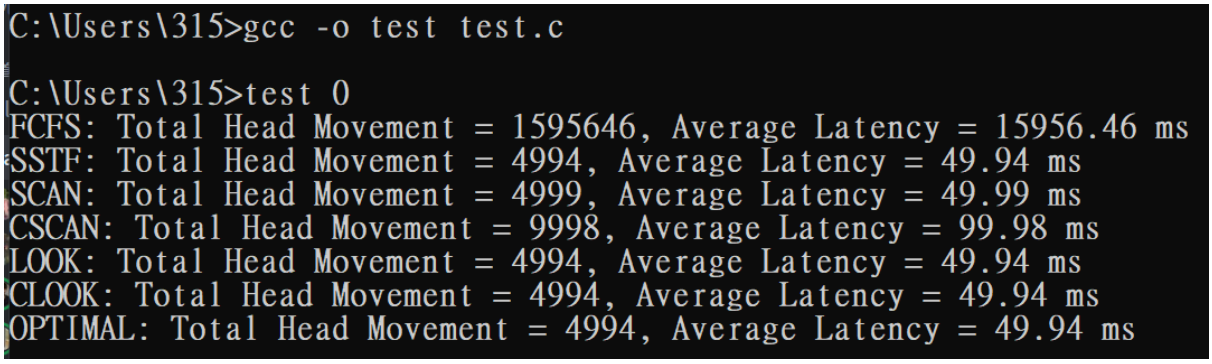
Now we change the position of head to 100.

```
C:\Users\315>test 100
FCFS: Total Head Movement = 1695977, Average Latency = 16959.77 ms
SSTF: Total Head Movement = 5087, Average Latency = 50.87 ms
SCAN: Total Head Movement = 9897, Average Latency = 98.97 ms
CSCAN: Total Head Movement = 9997, Average Latency = 99.97 ms
LOOK: Total Head Movement = 9877, Average Latency = 98.77 ms
CLOOK: Total Head Movement = 9975, Average Latency = 99.75 ms
OPTIMAL: Total Head Movement = 5087, Average Latency = 50.87 ms
```

Look at the graph above, we can know that SSTF is a great disk algorithm, it is very close to the optimal solution.

What is the total head movement required for each algorithm?

The total head movement is calculated within the functions. Running the provided program will give specific values for each algorithm.



FCFS has the largest value, so we can know that it is the simplest but worst solution of the disk algorithm.

Think about each algorithm, when would you use one algorithm versus another? List 2 applications where each algorithm would perform best.

1. FCFS

- Applications:
  - Simple systems where fairness is more critical than performance.
  - Systems with low request rates where complex scheduling is not necessary.

2. SSTF

- Applications:
  - Systems where minimizing seek time is critical, such as databases.
  - Real-time systems requiring quick access to nearby data.

3. SCAN

- Applications:
  - Systems with a heavy load of I/O operations where uniform wait time is important.
  - Multi-user systems where fair access to the disk is required.

4. CSCAN

- Applications:
  - Large-scale database systems where uniform wait time is crucial.
  - Real-time systems requiring predictable performance.

5. LOOK

- Applications:
  - Systems with variable load patterns where minimizing unnecessary movement is important.
  - Multi-user environments where efficiency is key.

6. CLOOK

- Applications:
  - Real-time systems needing both efficiency and uniform wait times.
  - Large-scale server systems where disk access patterns can be optimized.

7. OPTIMAL

- Applications:
  - Systems where all requests are known in advance, such as batch processing.
  - Simulation environments where optimal performance is required for testing.