9	Write a C program to implement the following contiguous memory allocation techniques : i) Worst-fit ii) Best-fit iii) First-fit

Memory Allocation Overview:

In operating systems, **contiguous memory allocation** is a method of assigning memory blocks to processes. It involves selecting memory blocks large enough to fit the process from available partitions. The goal is to **minimize fragmentation** and ensure efficient memory utilization.

---

1. First-Fit Algorithm:

Concept:

- The process is placed in the **first available block** that is large enough.
- **Fast and simple** but can lead to fragmentation over time.

Steps:

1. Start from the beginning of the memory blocks list.
2. Allocate the first block that can accommodate the process.
3. If no block fits, the process waits.

Time Complexity:

- **O(n) –** Linear scan through memory blocks.

---

2. Best-Fit Algorithm:

Concept:

- The process is placed in the **smallest available block** that fits.
- Minimizes leftover space but **can cause fragmentation** of small blocks.

Steps:

1. Scan all blocks and select the smallest block that can accommodate the process.
2. If no block fits, the process waits.

Time Complexity:

- **O(n) –** Linear scan to find the smallest fitting block.

---

3. Worst-Fit Algorithm:

Concept:

- The process is placed in the **largest available block**.

- Leaves the smallest unused blocks, reducing fragmentation initially but may lead to **large memory holes**.

Steps:

1. Scan all blocks and select the largest block that can accommodate the process.
2. If no block fits, the process waits.

Time Complexity:

- **O(n)** – Linear scan to find the largest fitting block.

Problem Example:

```
Memory Blocks: [100, 500, 200, 300, 600]
Processes: [212, 417, 112, 426]
```

Execution Flow (Sample Output):

```
Enter number of memory blocks: 5
Enter sizes of memory blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter number of processes: 4
Enter sizes of processes:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

First Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 5
Process 3 allocated to Block 2
Process 4 not allocated

Best Fit Allocation:
Process 1 allocated to Block 4
Process 2 allocated to Block 5
Process 3 allocated to Block 3
Process 4 not allocated

Worst Fit Allocation:
Process 1 allocated to Block 5
Process 2 allocated to Block 5
Process 3 allocated to Block 2
Process 4 not allocated
```

Analysis:

- **First-Fit:** Allocates quickly but leaves small unused blocks.
- **Best-Fit:** Utilizes space more efficiently but may leave fragmented memory blocks.
- **Worst-Fit:** Tries to avoid fragmentation but can lead to poor memory utilization over time.

Why These Algorithms Matter:

- **First-Fit** is **fast** and simple for dynamic memory allocation.
- **Best-Fit** reduces wasted memory but increases **search time**.
- **Worst-Fit** minimizes fragmentation initially but may **waste memory** in the long run.

```
#include <stdio.h>
#include <string.h>
#define MAX 10

void initializeAllocation(int allocation[], int size) {
    for (int i = 0; i < size; i++) allocation[i] = -1;
}

void displayAllocation(const char *method, int allocation[], int processes) {
    printf("\n%s Allocation:\n", method);
    for (int i = 0; i < processes; i++) {
        if (allocation[i] != -1)
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        else
            printf("Process %d not allocated\n", i + 1);
    }
}

void firstFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX];
    initializeAllocation(allocation, processes);

    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
    displayAllocation("First Fit", allocation, processes);
}

void bestFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX];
    initializeAllocation(allocation, processes);

    for (int i = 0; i < processes; i++) {
        int bestIdx = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }
    displayAllocation("Best Fit", allocation, processes);
}

void worstFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX];
    initializeAllocation(allocation, processes);

    for (int i = 0; i < processes; i++) {
        int worstIdx = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }
    displayAllocation("Worst Fit", allocation, processes);
}

int main() {
    int blockSize[MAX], processSize[MAX];
    int blocks, processes;

    printf("Enter number of memory blocks: ");
    scanf("%d", &blocks);
    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < blocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

    printf("\nEnter number of processes: ");
    scanf("%d", &processes);
    printf("Enter sizes of processes:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    int blockCopy[MAX];

    for (int i = 0; i < blocks; i++) blockCopy[i] = blockSize[i];
    firstFit(blockCopy, blocks, processSize, processes);

    for (int i = 0; i < blocks; i++) blockCopy[i] = blockSize[i];
    bestFit(blockCopy, blocks, processSize, processes);

    for (int i = 0; i < blocks; i++) blockCopy[i] = blockSize[i];
    worstFit(blockCopy, blocks, processSize, processes);

    return 0;
}
```

10	Develop a C program to implement the following page replacement algorithms: i) LRU ii) Optimal Also display the number of page faults encountered in each of the algorithm for a given reference string.

Page Replacement Overview:

Page replacement algorithms are used in operating systems to manage **virtual memory**. When a page fault occurs (i.e., a page is not found in memory), the system must replace one of the existing pages to bring in the required page. The goal is to **minimize page faults** by efficiently selecting which page to evict.

---

1. Least Recently Used (LRU) Algorithm:

Concept:

- LRU replaces the **least recently used** page when a new page needs to be loaded.
- It is based on the assumption that pages used recently will likely be used again soon.

Steps:

1. If the page is already in memory (frame), update its usage time.
2. If the page is not in memory:
    - If there is an empty frame, load the page.
    - If all frames are full, replace the page that was **least recently used**.

Time Complexity:

- **O(n × capacity)**, where `n` is the number of pages in the reference string and `capacity` is the number of frames.

---

2. Optimal Page Replacement Algorithm:

Concept:

- The **optimal algorithm** replaces the page that will **not be used for the longest period** in the future.
- This method guarantees the **least number of page faults** but is **theoretically impractical** because it requires knowledge of future requests.

Steps:

1. If the page is already in memory, do nothing.
2. If the page is not in memory:
    - If there is an empty frame, load the page.
    - If all frames are full, replace the page that **will not be used for the longest time in the future**.

Time Complexity:

- **O(n^2)**, due to scanning the future reference string for each page.

Problem Example:

```
Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3
Number of Frames: 3
```

Execution Flow (Sample Output):

```
Enter number of pages: 10
Enter the reference string: 7 0 1 2 0 3 0 4 2 3
Enter the number of frames: 3

First: LRU Simulation
7       -1      -1
7       0       -1
7       0       1
2       0       1
2       0       1
2       3       1
2       3       0
4       3       0
4       3       2
4       3       2

Total Page Faults (LRU) = 9

Next: Optimal Simulation
7       -1      -1
7       0       -1
7       0       1
2       0       1
2       0       1
3       0       1
3       0       4
3       4       2
```

```
3       4       2
3       4       2

Total Page Faults (Optimal) = 8
```

Analysis:

- **LRU Page Faults**: 9
- **Optimal Page Faults**: 8

Optimal has fewer page faults because it predicts future usage perfectly, while **LRU** only uses past access patterns.

Key Observations:

- **LRU** is easier to implement but may not perform as well as **Optimal** in all scenarios.
- **Optimal** provides the theoretical **best performance** but is not practical without future knowledge.

Why These Algorithms Matter:

- **LRU** is widely used in modern operating systems because it approximates real-world scenarios.
- **Optimal** is used as a benchmark to measure the performance of other algorithms.

```
#include <stdio.h>
#define MAX 20

int findLRU(int time[], int n) {
    int i, minimum = time[0], pos = 0;
    for (i = 1; i < n; ++i) {
        if (time[i] < minimum) {
            minimum = time[i];
            pos = i;
        }
    }
    return pos;
}

void lru(int pages[], int n, int capacity) {
    int frames[MAX], time[MAX], counter = 0, pageFaults = 0, i, j, pos;
    for (i = 0; i < capacity; ++i) {
        frames[i] = -1;
```

```c
    }

    for (i = 0; i < n; ++i) {
        int flag1 = 0, flag2 = 0;

        for (j = 0; j < capacity; ++j) {
            if (frames[j] == pages[i]) {
                counter++;
                time[j] = counter;
                flag1 = flag2 = 1;
                break;
            }
        }

        if (flag1 == 0) {
            for (j = 0; j < capacity; ++j) {
                if (frames[j] == -1) {
                    counter++;
                    pageFaults++;
                    frames[j] = pages[i];
                    time[j] = counter;
                    flag2 = 1;
                    break;
                }
            }
        }

        if (flag2 == 0) {
            pos = findLRU(time, capacity);
            counter++;
            pageFaults++;
            frames[pos] = pages[i];
            time[pos] = counter;
        }

        printf("\n");
        for (j = 0; j < capacity; ++j) {
            printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults (LRU) = %d\n", pageFaults);
}

void optimal(int pages[], int n, int capacity) {
    int frames[MAX], temp[MAX], pageFaults = 0;
    int i, j, k, pos, max;

    for (i = 0; i < capacity; ++i) {
        frames[i] = -1;
    }
```

```
for (i = 0; i < n; ++i) {
    int flag1 = 0, flag2 = 0;

    for (j = 0; j < capacity; ++j) {
        if (frames[j] == pages[i]) {
            flag1 = flag2 = 1;
            break;
        }
    }

    if (flag1 == 0) {
        for (j = 0; j < capacity; ++j) {
            if (frames[j] == -1) {
                pageFaults++;
                frames[j] = pages[i];
                flag2 = 1;
                break;
            }
        }
    }

    if (flag2 == 0) {
        for (j = 0; j < capacity; ++j) {
            temp[j] = -1;
            for (k = i + 1; k < n; ++k) {
                if (frames[j] == pages[k]) {
                    temp[j] = k;
                    break;
                }
            }
        }

        max = -1;
        for (j = 0; j < capacity; ++j) {
            if (temp[j] == -1) {
                pos = j;
                break;
            }
            if (temp[j] > max) {
                max = temp[j];
                pos = j;
            }
        }
        frames[pos] = pages[i];
        pageFaults++;
    }

    printf("\n");
    for (j = 0; j < capacity; ++j) {
        printf("%d\t", frames[j]);
    }
}
```

```
    printf("\n\nTotal Page Faults (Optimal) = %d\n", pageFaults);
}

int main() {
    int pages[MAX], n, capacity, i;

    printf("Enter number of pages: ");
    scanf("%d", &n);

    printf("Enter the reference string: ");
    for (i = 0; i < n; ++i) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &capacity);

    lru(pages, n, capacity);
    optimal(pages, n, capacity);

    return 0;
}
```