Christian Schuering

Assignment 1

Programs 1 – 7 and Documentation

Program 1 –

```
#include <iostream>

using namespace std;

void move(int disk, char source, char dest, int &count);
void Hanoi(int disk, char Start, char A1, char A2, char A3, char A4, char A5,
char Dest, int &count);
void HanoiS_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count);
void HanoiA5_to_A3(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count);
void HanoiA2_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count);
void HanoiA3_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count);
void HanoiA4_to_D(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count);
void HanoiFin(int disk, char Start, char A1, char A2, char A3, char A4, char A5,
char Dest, int &count);

void move(int disk, char source, char dest, int &count) {
    count++;
    if (count <= 100 || count > count - 100) {
        cout << "Move " << count << ": move disk " << disk << " from " << source
<< " to " << dest << endl;
    }
}
//initializes k - 1 disks and continues to begin the tower
void Hanoi(int disk, char Start, char A1, char A2, char A3, char A4, char A5,
char Dest, int &count){
    if(disk == 1){
        move(disk, Start, A1, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
        move(disk, A3, A4, count);
        move(disk, A4, Dest, count);
    }else if(disk >= 2){
        HanoiS_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
```

```
        move(disk, Start, A1, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
        move(disk, A3, A4, count);
        HanoiA4_to_D(disk, Start, A1, A2, A3, A4, A5, Dest, count);
        HanoiFin(disk, Start, A1, A2, A3, A4, A5, Dest, count);
    }
}

//Starts the original disk which starts n = 1
void HanoiS_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count){
    if(disk == 1){
        move(disk, Start, A1, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
        move(disk, A3, A4, count);
        move(disk, A4, A5, count);
    }else if(disk >= 2){
        HanoiS_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        move(disk, Start, A1, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
        move(disk, A3, A4, count);
        HanoiA5_to_A3(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        move(disk, A4, A5, count);
        //HanoiA2_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        HanoiA3_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
    }
} //for n = 4, move 24 must be move disk 3 from 4 to 5

//Moves disk where k - 1 == n faster to the auxiliary which is 3
void HanoiA5_to_A3(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count){
    if (disk == 1){
        move(disk, A5, A1, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
    }else if(disk >= 2){
        HanoiA5_to_A3(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        move(disk, A5, A1, count);
        HanoiA3_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        move(disk, A1, A2, count);
        move(disk, A2, A3, count);
        move(disk, A3, A4, count);
```

```
        //HanoiA4_to_D(disk, Start, A1, A2, A3, A4, A5, Dest, count);
        //move(disk, A5, A1, count);
        //move(disk, A1, A2, count);
    }
}
//presents option to go deeper as well as move disks to peg #5
void HanoiA2_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count){
    move(disk, A2, A3, count);
    move(disk, A3, A4, count);
    move(disk, A4, A5, count);
    HanoiA3_to_A5(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
}
//presents faster option to have smaller disks go to peg #5 faster
void HanoiA3_to_A5(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count){ //fix
    if(disk == 1){
        move(disk, A3, A4, count);
        move(disk, A4, A5, count);
    }else if(disk >= 2){
        move(disk, A3, A4, count);
        move(disk, A4, A5, count);
    }
}
//helps with destination tracking and pushing forward into the k-1 disk
void HanoiA4_to_D(int disk, char Start, char A1, char A2, char A3, char A4, char
A5, char Dest, int &count){
    if(disk == 1){
        move(disk, A3, A4, count);
        move(disk, A4, Dest, count);
    }else if(disk == 2){
        HanoiA5_to_A3(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
        move(disk, A4, A5, count);
    }else if(disk >= 3){
        move(disk, A4, Dest, count);
        HanoiA5_to_A3(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
    }
}
    //exit code to follow through with destinations
 void HanoiFin(int disk, char Start, char A1, char A2, char A3, char A4, char A5,
char Dest, int &count){
    if(disk <= 0) return;
    if(disk == 1){
        move(disk, A3, A4, count);
        move(disk, A4, Dest, count);
```

```
    }else if(disk == 2){
        //move(disk, A3, A4, count);
        move(disk, A4, Dest, count);
    }
    HanoiA5_to_A3(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
    HanoiFin(disk-1, Start, A1, A2, A3, A4, A5, Dest, count);
}

int main(){
    int n = 0; //test how many disks you'd like to play with
    int count = 0;

    cout << "Please enter how many disks we will be playing with today: " <<
endl;
    cin >> n;

    Hanoi(n, 'S', 'A1', 'A2', 'A3', 'A4', 'A5', 'D', count); //Edges in row
    cout << "Total moves for the disk was: "<< count << endl;
    return 0;
}

//valid moves
// S -> A1 -> A2 -> A3 -> A4 -> A5 -> A1 -> A4 -> D
//all moves will be 7 steps at least
//no H4->D disk-1

/*
HYPOTHESIS - I believe the amount of moves, given the material we've studied will
skyrocket
given that the time complexity is O(2^n). I would say that n = 10 will result in
more than
50,000 moves.

This code provides a solver for the classic Towers of Hanoi problem. The Towers
of Hanoi
is a mathematical puzzle that involves moving a stack of disks from one tower to
another,
using auxiliary towers, with the constraint that a larger disk must never be
placed on
top of a smaller disk.
The code is implemented in C++ and follows a recursive approach to solve the
problem.
It utilizes several helper functions to break down the problem into smaller
subproblems
and perform the necessary moves.
```

The main function, main(), prompts the user to enter the number of disks they want to play with. It then calls the Hanoi() function with the specified number of disks and the tower labels ('S', 'A1', 'A2', 'A3', 'A4', 'A5', 'D') representing the source tower, auxiliary towers, and destination tower, respectively. The function also initializes a counter variable count to keep track of the total number of moves.

The Hanoi() function is the primary recursive function that solves the Towers of Hanoi problem. It takes the number of disks, the tower labels, and the count as parameters.

The function is divided into base cases and recursive cases.

In the base case (disk == 1), the function directly moves the disk from the source tower to the destination tower using a series of move() function calls. The move() function increments the count and prints the details of the move.

In the recursive cases (disk >= 2), the function follows a specific sequence of moves to solve the problem. It first recursively calls HanoiS_to_A5() to move disk-1 disks from the source tower to the auxiliary tower A5. Then it performs a series of moves to move the largest disk from the source tower to the destination tower via the auxiliary towers A1, A2, A3, and A4. After that, it recursively calls HanoiA4_to_D() to move the remaining disks from A4 to the destination tower. Finally, it calls HanoiFin() to handle the remaining disks and complete the solution.

The other helper functions (HanoiS_to_A5(), HanoiA5_to_A3(), HanoiA2_to_A5(), HanoiA3_to_A5(), HanoiA4_to_D(), HanoiFin()) are used to handle specific move sequences within the recursive cases of Hanoi().

At the end of the main() function, the total number of moves (count) is printed as the output.

Time Complexity:

The time complexity of the code is determined by the number of moves required to solve the Towers of Hanoi problem for a given number of disks (n). The time complexity

can be represented as O(2^n), where n is the number of disks. This is because each
disk can be moved twice (once from the source to an auxiliary tower and once from the auxiliary tower to the destination) and there are 2^n - 1 total moves required
to solve the problem.

Space Complexity:
The space complexity of the code is O(1). It does not utilize any additional data structures that grow with the input size (n). The space required is constant, as it only uses a few variables to keep track of the state of the algorithm (such as disk, source, dest, and count). The space complexity does not depend on the number
of disks being used.

```
Please enter how many disks we will be playing with today:
1
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to D
Total moves for the disk was: 5

Please enter how many disks we will be playing with today:
2
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 2 from 4 to D
Move 15: move disk 1 from 5 to 1
Move 16: move disk 1 from 1 to 2
Move 17: move disk 1 from 2 to 3
Move 18: move disk 1 from 3 to 4
Move 19: move disk 1 from 4 to D
```

```
Total moves for the disk was: 19

Please enter how many disks we will be playing with today:
3
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 3 from 4 to D
Move 21: move disk 1 from 5 to 1
Move 22: move disk 1 from 1 to 2
Move 23: move disk 1 from 2 to 3
Move 24: move disk 2 from 5 to 1
Move 25: move disk 1 from 3 to 4
Move 26: move disk 1 from 4 to 5
Move 27: move disk 2 from 1 to 2
Move 28: move disk 2 from 2 to 3
Move 29: move disk 2 from 3 to 4
Move 30: move disk 1 from 5 to 1
Move 31: move disk 1 from 1 to 2
Move 32: move disk 1 from 2 to 3
Move 33: move disk 2 from 5 to 1
Move 34: move disk 1 from 3 to 4
Move 35: move disk 1 from 4 to 5
Move 36: move disk 2 from 1 to 2
Move 37: move disk 2 from 2 to 3
Move 38: move disk 2 from 3 to 4
Move 39: move disk 2 from 4 to D
Move 40: move disk 1 from 5 to 1
Move 41: move disk 1 from 1 to 2
```

```
Move 42: move disk 1 from 2 to 3
Move 43: move disk 1 from 3 to 4
Move 44: move disk 1 from 4 to D
Total moves for the disk was: 44

Please enter how many disks we will be playing with today:
4
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 4 from 4 to D
Move 37: move disk 1 from 5 to 1
Move 38: move disk 1 from 1 to 2
```

```
Move 39: move disk 1 from 2 to 3
Move 40: move disk 2 from 5 to 1
Move 41: move disk 1 from 3 to 4
Move 42: move disk 1 from 4 to 5
Move 43: move disk 2 from 1 to 2
Move 44: move disk 2 from 2 to 3
Move 45: move disk 2 from 3 to 4
Move 46: move disk 3 from 5 to 1
Move 47: move disk 2 from 3 to 4
Move 48: move disk 2 from 4 to 5
Move 49: move disk 3 from 1 to 2
Move 50: move disk 3 from 2 to 3
Move 51: move disk 3 from 3 to 4
Move 52: move disk 1 from 5 to 1
Move 53: move disk 1 from 1 to 2
Move 54: move disk 1 from 2 to 3
Move 55: move disk 2 from 5 to 1
Move 56: move disk 1 from 3 to 4
Move 57: move disk 1 from 4 to 5
Move 58: move disk 2 from 1 to 2
Move 59: move disk 2 from 2 to 3
Move 60: move disk 2 from 3 to 4
Move 61: move disk 3 from 5 to 1
Move 62: move disk 2 from 3 to 4
Move 63: move disk 2 from 4 to 5
Move 64: move disk 3 from 1 to 2
Move 65: move disk 3 from 2 to 3
Move 66: move disk 3 from 3 to 4
Move 67: move disk 1 from 5 to 1
Move 68: move disk 1 from 1 to 2
Move 69: move disk 1 from 2 to 3
Move 70: move disk 2 from 5 to 1
Move 71: move disk 1 from 3 to 4
Move 72: move disk 1 from 4 to 5
Move 73: move disk 2 from 1 to 2
Move 74: move disk 2 from 2 to 3
Move 75: move disk 2 from 3 to 4
Move 76: move disk 2 from 4 to D
Move 77: move disk 1 from 5 to 1
Move 78: move disk 1 from 1 to 2
Move 79: move disk 1 from 2 to 3
Move 80: move disk 1 from 3 to 4
Move 81: move disk 1 from 4 to D
Total moves for the disk was: 81
```

```
Please enter how many disks we will be playing with today:
5
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
Move 43: move disk 2 from 2 to 3
```

```
Move 44: move disk 2 from 3 to 4
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 5 from 4 to D
Move 59: move disk 1 from 5 to 1
Move 60: move disk 1 from 1 to 2
Move 61: move disk 1 from 2 to 3
Move 62: move disk 2 from 5 to 1
Move 63: move disk 1 from 3 to 4
Move 64: move disk 1 from 4 to 5
Move 65: move disk 2 from 1 to 2
Move 66: move disk 2 from 2 to 3
Move 67: move disk 2 from 3 to 4
Move 68: move disk 3 from 5 to 1
Move 69: move disk 2 from 3 to 4
Move 70: move disk 2 from 4 to 5
Move 71: move disk 3 from 1 to 2
Move 72: move disk 3 from 2 to 3
Move 73: move disk 3 from 3 to 4
Move 74: move disk 4 from 5 to 1
Move 75: move disk 3 from 3 to 4
Move 76: move disk 3 from 4 to 5
Move 77: move disk 4 from 1 to 2
Move 78: move disk 4 from 2 to 3
Move 79: move disk 4 from 3 to 4
Move 80: move disk 1 from 5 to 1
Move 81: move disk 1 from 1 to 2
Move 82: move disk 1 from 2 to 3
Move 83: move disk 2 from 5 to 1
Move 84: move disk 1 from 3 to 4
Move 85: move disk 1 from 4 to 5
Move 86: move disk 2 from 1 to 2
Move 87: move disk 2 from 2 to 3
Move 88: move disk 2 from 3 to 4
```

```
Move 89: move disk 3 from 5 to 1
Move 90: move disk 2 from 3 to 4
Move 91: move disk 2 from 4 to 5
Move 92: move disk 3 from 1 to 2
Move 93: move disk 3 from 2 to 3
Move 94: move disk 3 from 3 to 4
Move 95: move disk 4 from 5 to 1
Move 96: move disk 3 from 3 to 4
Move 97: move disk 3 from 4 to 5
Move 98: move disk 4 from 1 to 2
Move 99: move disk 4 from 2 to 3
Move 100: move disk 4 from 3 to 4
Move 101: move disk 1 from 5 to 1
Move 102: move disk 1 from 1 to 2
Move 103: move disk 1 from 2 to 3
Move 104: move disk 2 from 5 to 1
Move 105: move disk 1 from 3 to 4
Move 106: move disk 1 from 4 to 5
Move 107: move disk 2 from 1 to 2
Move 108: move disk 2 from 2 to 3
Move 109: move disk 2 from 3 to 4
Move 110: move disk 3 from 5 to 1
Move 111: move disk 2 from 3 to 4
Move 112: move disk 2 from 4 to 5
Move 113: move disk 3 from 1 to 2
Move 114: move disk 3 from 2 to 3
Move 115: move disk 3 from 3 to 4
Move 116: move disk 1 from 5 to 1
Move 117: move disk 1 from 1 to 2
Move 118: move disk 1 from 2 to 3
Move 119: move disk 2 from 5 to 1
Move 120: move disk 1 from 3 to 4
Move 121: move disk 1 from 4 to 5
Move 122: move disk 2 from 1 to 2
Move 123: move disk 2 from 2 to 3
Move 124: move disk 2 from 3 to 4
Move 125: move disk 2 from 4 to D
Move 126: move disk 1 from 5 to 1
Move 127: move disk 1 from 1 to 2
Move 128: move disk 1 from 2 to 3
Move 129: move disk 1 from 3 to 4
Move 130: move disk 1 from 4 to D
Total moves for the disk was: 130

Please enter how many disks we will be playing with today:
```

```
6
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
Move 43: move disk 2 from 2 to 3
Move 44: move disk 2 from 3 to 4
```

```
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 1 from 5 to 1
Move 59: move disk 1 from 1 to 2
Move 60: move disk 1 from 2 to 3
Move 61: move disk 2 from 5 to 1
Move 62: move disk 1 from 3 to 4
Move 63: move disk 1 from 4 to 5
Move 64: move disk 2 from 1 to 2
Move 65: move disk 2 from 2 to 3
Move 66: move disk 2 from 3 to 4
Move 67: move disk 3 from 5 to 1
Move 68: move disk 2 from 3 to 4
Move 69: move disk 2 from 4 to 5
Move 70: move disk 3 from 1 to 2
Move 71: move disk 3 from 2 to 3
Move 72: move disk 3 from 3 to 4
Move 73: move disk 4 from 5 to 1
Move 74: move disk 3 from 3 to 4
Move 75: move disk 3 from 4 to 5
Move 76: move disk 4 from 1 to 2
Move 77: move disk 4 from 2 to 3
Move 78: move disk 4 from 3 to 4
Move 79: move disk 5 from 4 to 5
Move 80: move disk 4 from 3 to 4
Move 81: move disk 4 from 4 to 5
Move 82: move disk 6 from S to 1
Move 83: move disk 6 from 1 to 2
Move 84: move disk 6 from 2 to 3
Move 85: move disk 6 from 3 to 4
Move 86: move disk 6 from 4 to D
Move 87: move disk 1 from 5 to 1
Move 88: move disk 1 from 1 to 2
Move 89: move disk 1 from 2 to 3
```

```
Move 90: move disk 2 from 5 to 1
Move 91: move disk 1 from 3 to 4
Move 92: move disk 1 from 4 to 5
Move 93: move disk 2 from 1 to 2
Move 94: move disk 2 from 2 to 3
Move 95: move disk 2 from 3 to 4
Move 96: move disk 3 from 5 to 1
Move 97: move disk 2 from 3 to 4
Move 98: move disk 2 from 4 to 5
Move 99: move disk 3 from 1 to 2
Move 100: move disk 3 from 2 to 3
Move 101: move disk 3 from 3 to 4
Move 102: move disk 4 from 5 to 1
Move 103: move disk 3 from 3 to 4
Move 104: move disk 3 from 4 to 5
Move 105: move disk 4 from 1 to 2
Move 106: move disk 4 from 2 to 3
Move 107: move disk 4 from 3 to 4
Move 108: move disk 5 from 5 to 1
Move 109: move disk 4 from 3 to 4
Move 110: move disk 4 from 4 to 5
Move 111: move disk 5 from 1 to 2
Move 112: move disk 5 from 2 to 3
Move 113: move disk 5 from 3 to 4
Move 114: move disk 1 from 5 to 1
Move 115: move disk 1 from 1 to 2
Move 116: move disk 1 from 2 to 3
Move 117: move disk 2 from 5 to 1
Move 118: move disk 1 from 3 to 4
Move 119: move disk 1 from 4 to 5
Move 120: move disk 2 from 1 to 2
Move 121: move disk 2 from 2 to 3
Move 122: move disk 2 from 3 to 4
Move 123: move disk 3 from 5 to 1
Move 124: move disk 2 from 3 to 4
Move 125: move disk 2 from 4 to 5
Move 126: move disk 3 from 1 to 2
Move 127: move disk 3 from 2 to 3
Move 128: move disk 3 from 3 to 4
Move 129: move disk 4 from 5 to 1
Move 130: move disk 3 from 3 to 4
Move 131: move disk 3 from 4 to 5
Move 132: move disk 4 from 1 to 2
Move 133: move disk 4 from 2 to 3
Move 134: move disk 4 from 3 to 4
```

```
Move 135: move disk 5 from 5 to 1
Move 136: move disk 4 from 3 to 4
Move 137: move disk 4 from 4 to 5
Move 138: move disk 5 from 1 to 2
Move 139: move disk 5 from 2 to 3
Move 140: move disk 5 from 3 to 4
Move 141: move disk 1 from 5 to 1
Move 142: move disk 1 from 1 to 2
Move 143: move disk 1 from 2 to 3
Move 144: move disk 2 from 5 to 1
Move 145: move disk 1 from 3 to 4
Move 146: move disk 1 from 4 to 5
Move 147: move disk 2 from 1 to 2
Move 148: move disk 2 from 2 to 3
Move 149: move disk 2 from 3 to 4
Move 150: move disk 3 from 5 to 1
Move 151: move disk 2 from 3 to 4
Move 152: move disk 2 from 4 to 5
Move 153: move disk 3 from 1 to 2
Move 154: move disk 3 from 2 to 3
Move 155: move disk 3 from 3 to 4
Move 156: move disk 4 from 5 to 1
Move 157: move disk 3 from 3 to 4
Move 158: move disk 3 from 4 to 5
Move 159: move disk 4 from 1 to 2
Move 160: move disk 4 from 2 to 3
Move 161: move disk 4 from 3 to 4
Move 162: move disk 1 from 5 to 1
Move 163: move disk 1 from 1 to 2
Move 164: move disk 1 from 2 to 3
Move 165: move disk 2 from 5 to 1
Move 166: move disk 1 from 3 to 4
Move 167: move disk 1 from 4 to 5
Move 168: move disk 2 from 1 to 2
Move 169: move disk 2 from 2 to 3
Move 170: move disk 2 from 3 to 4
Move 171: move disk 3 from 5 to 1
Move 172: move disk 2 from 3 to 4
Move 173: move disk 2 from 4 to 5
Move 174: move disk 3 from 1 to 2
Move 175: move disk 3 from 2 to 3
Move 176: move disk 3 from 3 to 4
Move 177: move disk 1 from 5 to 1
Move 178: move disk 1 from 1 to 2
Move 179: move disk 1 from 2 to 3
```

```
Move 180: move disk 2 from 5 to 1
Move 181: move disk 1 from 3 to 4
Move 182: move disk 1 from 4 to 5
Move 183: move disk 2 from 1 to 2
Move 184: move disk 2 from 2 to 3
Move 185: move disk 2 from 3 to 4
Move 186: move disk 2 from 4 to D
Move 187: move disk 1 from 5 to 1
Move 188: move disk 1 from 1 to 2
Move 189: move disk 1 from 2 to 3
Move 190: move disk 1 from 3 to 4
Move 191: move disk 1 from 4 to D
Total moves for the disk was: 191

Please enter how many disks we will be playing with today:
7
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
```

```
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
Move 43: move disk 2 from 2 to 3
Move 44: move disk 2 from 3 to 4
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 1 from 5 to 1
Move 59: move disk 1 from 1 to 2
Move 60: move disk 1 from 2 to 3
Move 61: move disk 2 from 5 to 1
Move 62: move disk 1 from 3 to 4
Move 63: move disk 1 from 4 to 5
Move 64: move disk 2 from 1 to 2
Move 65: move disk 2 from 2 to 3
Move 66: move disk 2 from 3 to 4
Move 67: move disk 3 from 5 to 1
Move 68: move disk 2 from 3 to 4
Move 69: move disk 2 from 4 to 5
Move 70: move disk 3 from 1 to 2
Move 71: move disk 3 from 2 to 3
Move 72: move disk 3 from 3 to 4
Move 73: move disk 4 from 5 to 1
Move 74: move disk 3 from 3 to 4
```

```
Move 75: move disk 3 from 4 to 5
Move 76: move disk 4 from 1 to 2
Move 77: move disk 4 from 2 to 3
Move 78: move disk 4 from 3 to 4
Move 79: move disk 5 from 4 to 5
Move 80: move disk 4 from 3 to 4
Move 81: move disk 4 from 4 to 5
Move 82: move disk 6 from S to 1
Move 83: move disk 6 from 1 to 2
Move 84: move disk 6 from 2 to 3
Move 85: move disk 6 from 3 to 4
Move 86: move disk 1 from 5 to 1
Move 87: move disk 1 from 1 to 2
Move 88: move disk 1 from 2 to 3
Move 89: move disk 2 from 5 to 1
Move 90: move disk 1 from 3 to 4
Move 91: move disk 1 from 4 to 5
Move 92: move disk 2 from 1 to 2
Move 93: move disk 2 from 2 to 3
Move 94: move disk 2 from 3 to 4
Move 95: move disk 3 from 5 to 1
Move 96: move disk 2 from 3 to 4
Move 97: move disk 2 from 4 to 5
Move 98: move disk 3 from 1 to 2
Move 99: move disk 3 from 2 to 3
Move 100: move disk 3 from 3 to 4
... (Could not get to work so this was done by hand)
Move 164: move disk 2 from 3 to 4
Move 165: move disk 2 from 4 to 5
Move 166: move disk 3 from 1 to 2
Move 167: move disk 3 from 2 to 3
Move 168: move disk 3 from 3 to 4
Move 169: move disk 4 from 5 to 1
Move 170: move disk 3 from 3 to 4
Move 171: move disk 3 from 4 to 5
Move 172: move disk 4 from 1 to 2
Move 173: move disk 4 from 2 to 3
Move 174: move disk 4 from 3 to 4
Move 175: move disk 5 from 5 to 1
Move 176: move disk 4 from 3 to 4
Move 177: move disk 4 from 4 to 5
Move 178: move disk 5 from 1 to 2
Move 179: move disk 5 from 2 to 3
Move 180: move disk 5 from 3 to 4
Move 181: move disk 6 from 5 to 1
```

Move 182: move disk 5 from 3 to 4
Move 183: move disk 5 from 4 to 5
Move 184: move disk 6 from 1 to 2
Move 185: move disk 6 from 2 to 3
Move 186: move disk 6 from 3 to 4
Move 187: move disk 1 from 5 to 1
Move 188: move disk 1 from 1 to 2
Move 189: move disk 1 from 2 to 3
Move 190: move disk 2 from 5 to 1
Move 191: move disk 1 from 3 to 4
Move 192: move disk 1 from 4 to 5
Move 193: move disk 2 from 1 to 2
Move 194: move disk 2 from 2 to 3
Move 195: move disk 2 from 3 to 4
Move 196: move disk 3 from 5 to 1
Move 197: move disk 2 from 3 to 4
Move 198: move disk 2 from 4 to 5
Move 199: move disk 3 from 1 to 2
Move 200: move disk 3 from 2 to 3
Move 201: move disk 3 from 3 to 4
Move 202: move disk 4 from 5 to 1
Move 203: move disk 3 from 3 to 4
Move 204: move disk 3 from 4 to 5
Move 205: move disk 4 from 1 to 2
Move 206: move disk 4 from 2 to 3
Move 207: move disk 4 from 3 to 4
Move 208: move disk 5 from 5 to 1
Move 209: move disk 4 from 3 to 4
Move 210: move disk 4 from 4 to 5
Move 211: move disk 5 from 1 to 2
Move 212: move disk 5 from 2 to 3
Move 213: move disk 5 from 3 to 4
Move 214: move disk 1 from 5 to 1
Move 215: move disk 1 from 1 to 2
Move 216: move disk 1 from 2 to 3
Move 217: move disk 2 from 5 to 1
Move 218: move disk 1 from 3 to 4
Move 219: move disk 1 from 4 to 5
Move 220: move disk 2 from 1 to 2
Move 221: move disk 2 from 2 to 3
Move 222: move disk 2 from 3 to 4
Move 223: move disk 3 from 5 to 1
Move 224: move disk 2 from 3 to 4
Move 225: move disk 2 from 4 to 5
Move 226: move disk 3 from 1 to 2

```
Move 227: move disk 3 from 2 to 3
Move 228: move disk 3 from 3 to 4
Move 229: move disk 4 from 5 to 1
Move 230: move disk 3 from 3 to 4
Move 231: move disk 3 from 4 to 5
Move 232: move disk 4 from 1 to 2
Move 233: move disk 4 from 2 to 3
Move 234: move disk 4 from 3 to 4
Move 235: move disk 1 from 5 to 1
Move 236: move disk 1 from 1 to 2
Move 237: move disk 1 from 2 to 3
Move 238: move disk 2 from 5 to 1
Move 239: move disk 1 from 3 to 4
Move 240: move disk 1 from 4 to 5
Move 241: move disk 2 from 1 to 2
Move 242: move disk 2 from 2 to 3
Move 243: move disk 2 from 3 to 4
Move 244: move disk 3 from 5 to 1
Move 245: move disk 2 from 3 to 4
Move 246: move disk 2 from 4 to 5
Move 247: move disk 3 from 1 to 2
Move 248: move disk 3 from 2 to 3
Move 249: move disk 3 from 3 to 4
Move 250: move disk 1 from 5 to 1
Move 251: move disk 1 from 1 to 2
Move 252: move disk 1 from 2 to 3
Move 253: move disk 2 from 5 to 1
Move 254: move disk 1 from 3 to 4
Move 255: move disk 1 from 4 to 5
Move 256: move disk 2 from 1 to 2
Move 257: move disk 2 from 2 to 3
Move 258: move disk 2 from 3 to 4
Move 259: move disk 2 from 4 to D
Move 260: move disk 1 from 5 to 1
Move 261: move disk 1 from 1 to 2
Move 262: move disk 1 from 2 to 3
Move 263: move disk 1 from 3 to 4
Move 264: move disk 1 from 4 to D

Please enter how many disks we will be playing with today:
8
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
```

```
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
Move 43: move disk 2 from 2 to 3
Move 44: move disk 2 from 3 to 4
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
```

```
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 1 from 5 to 1
Move 59: move disk 1 from 1 to 2
Move 60: move disk 1 from 2 to 3
Move 61: move disk 2 from 5 to 1
Move 62: move disk 1 from 3 to 4
Move 63: move disk 1 from 4 to 5
Move 64: move disk 2 from 1 to 2
Move 65: move disk 2 from 2 to 3
Move 66: move disk 2 from 3 to 4
Move 67: move disk 3 from 5 to 1
Move 68: move disk 2 from 3 to 4
Move 69: move disk 2 from 4 to 5
Move 70: move disk 3 from 1 to 2
Move 71: move disk 3 from 2 to 3
Move 72: move disk 3 from 3 to 4
Move 73: move disk 4 from 5 to 1
Move 74: move disk 3 from 3 to 4
Move 75: move disk 3 from 4 to 5
Move 76: move disk 4 from 1 to 2
Move 77: move disk 4 from 2 to 3
Move 78: move disk 4 from 3 to 4
Move 79: move disk 5 from 4 to 5
Move 80: move disk 4 from 3 to 4
Move 81: move disk 4 from 4 to 5
Move 82: move disk 6 from S to 1
Move 83: move disk 6 from 1 to 2
Move 84: move disk 6 from 2 to 3
Move 85: move disk 6 from 3 to 4
Move 86: move disk 1 from 5 to 1
Move 87: move disk 1 from 1 to 2
Move 88: move disk 1 from 2 to 3
Move 89: move disk 2 from 5 to 1
Move 90: move disk 1 from 3 to 4
Move 91: move disk 1 from 4 to 5
Move 92: move disk 2 from 1 to 2
Move 93: move disk 2 from 2 to 3
Move 94: move disk 2 from 3 to 4
```

```
Move 95: move disk 3 from 5 to 1
Move 96: move disk 2 from 3 to 4
Move 97: move disk 2 from 4 to 5
Move 98: move disk 3 from 1 to 2
Move 99: move disk 3 from 2 to 3
Move 100: move disk 3 from 3 to 4
... (Could not get to work so this was done by hand)
Move 249: move disk 2 from 3 to 4
Move 250: move disk 2 from 4 to 5
Move 251: move disk 3 from 1 to 2
Move 252: move disk 3 from 2 to 3
Move 253: move disk 3 from 3 to 4
Move 254: move disk 4 from 5 to 1
Move 255: move disk 3 from 3 to 4
Move 256: move disk 3 from 4 to 5
Move 257: move disk 4 from 1 to 2
Move 258: move disk 4 from 2 to 3
Move 259: move disk 4 from 3 to 4
Move 260: move disk 5 from 5 to 1
Move 261: move disk 4 from 3 to 4
Move 262: move disk 4 from 4 to 5
Move 263: move disk 5 from 1 to 2
Move 264: move disk 5 from 2 to 3
Move 265: move disk 5 from 3 to 4
Move 266: move disk 6 from 5 to 1
Move 267: move disk 5 from 3 to 4
Move 268: move disk 5 from 4 to 5
Move 269: move disk 6 from 1 to 2
Move 270: move disk 6 from 2 to 3
Move 271: move disk 6 from 3 to 4
Move 272: move disk 1 from 5 to 1
Move 273: move disk 1 from 1 to 2
Move 274: move disk 1 from 2 to 3
Move 275: move disk 2 from 5 to 1
Move 276: move disk 1 from 3 to 4
Move 277: move disk 1 from 4 to 5
Move 278: move disk 2 from 1 to 2
Move 279: move disk 2 from 2 to 3
Move 280: move disk 2 from 3 to 4
Move 281: move disk 3 from 5 to 1
Move 282: move disk 2 from 3 to 4
Move 283: move disk 2 from 4 to 5
Move 284: move disk 3 from 1 to 2
Move 285: move disk 3 from 2 to 3
Move 286: move disk 3 from 3 to 4
```

Move 287: move disk 4 from 5 to 1
Move 288: move disk 3 from 3 to 4
Move 289: move disk 3 from 4 to 5
Move 290: move disk 4 from 1 to 2
Move 291: move disk 4 from 2 to 3
Move 292: move disk 4 from 3 to 4
Move 293: move disk 5 from 5 to 1
Move 294: move disk 4 from 3 to 4
Move 295: move disk 4 from 4 to 5
Move 296: move disk 5 from 1 to 2
Move 297: move disk 5 from 2 to 3
Move 298: move disk 5 from 3 to 4
Move 299: move disk 1 from 5 to 1
Move 300: move disk 1 from 1 to 2
Move 301: move disk 1 from 2 to 3
Move 302: move disk 2 from 5 to 1
Move 303: move disk 1 from 3 to 4
Move 304: move disk 1 from 4 to 5
Move 305: move disk 2 from 1 to 2
Move 306: move disk 2 from 2 to 3
Move 307: move disk 2 from 3 to 4
Move 308: move disk 3 from 5 to 1
Move 309: move disk 2 from 3 to 4
Move 310: move disk 2 from 4 to 5
Move 311: move disk 3 from 1 to 2
Move 312: move disk 3 from 2 to 3
Move 313: move disk 3 from 3 to 4
Move 314: move disk 4 from 5 to 1
Move 315: move disk 3 from 3 to 4
Move 316: move disk 3 from 4 to 5
Move 317: move disk 4 from 1 to 2
Move 318: move disk 4 from 2 to 3
Move 319: move disk 4 from 3 to 4
Move 320: move disk 1 from 5 to 1
Move 321: move disk 1 from 1 to 2
Move 322: move disk 1 from 2 to 3
Move 323: move disk 2 from 5 to 1
Move 324: move disk 1 from 3 to 4
Move 325: move disk 1 from 4 to 5
Move 326: move disk 2 from 1 to 2
Move 327: move disk 2 from 2 to 3
Move 328: move disk 2 from 3 to 4
Move 329: move disk 3 from 5 to 1
Move 330: move disk 2 from 3 to 4
Move 331: move disk 2 from 4 to 5

```
Move 332: move disk 3 from 1 to 2
Move 333: move disk 3 from 2 to 3
Move 334: move disk 3 from 3 to 4
Move 335: move disk 1 from 5 to 1
Move 336: move disk 1 from 1 to 2
Move 337: move disk 1 from 2 to 3
Move 338: move disk 2 from 5 to 1
Move 339: move disk 1 from 3 to 4
Move 340: move disk 1 from 4 to 5
Move 341: move disk 2 from 1 to 2
Move 342: move disk 2 from 2 to 3
Move 343: move disk 2 from 3 to 4
Move 344: move disk 2 from 4 to D
Move 345: move disk 1 from 5 to 1
Move 346: move disk 1 from 1 to 2
Move 347: move disk 1 from 2 to 3
Move 348: move disk 1 from 3 to 4
Move 349: move disk 1 from 4 to D
Total moves for the disk was: 349

Please enter how many disks we will be playing with today:
9
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
```

```
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
Move 43: move disk 2 from 2 to 3
Move 44: move disk 2 from 3 to 4
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 1 from 5 to 1
Move 59: move disk 1 from 1 to 2
Move 60: move disk 1 from 2 to 3
Move 61: move disk 2 from 5 to 1
Move 62: move disk 1 from 3 to 4
Move 63: move disk 1 from 4 to 5
Move 64: move disk 2 from 1 to 2
Move 65: move disk 2 from 2 to 3
Move 66: move disk 2 from 3 to 4
Move 67: move disk 3 from 5 to 1
Move 68: move disk 2 from 3 to 4
```

```
Move 69: move disk 2 from 4 to 5
Move 70: move disk 3 from 1 to 2
Move 71: move disk 3 from 2 to 3
Move 72: move disk 3 from 3 to 4
Move 73: move disk 4 from 5 to 1
Move 74: move disk 3 from 3 to 4
Move 75: move disk 3 from 4 to 5
Move 76: move disk 4 from 1 to 2
Move 77: move disk 4 from 2 to 3
Move 78: move disk 4 from 3 to 4
Move 79: move disk 5 from 4 to 5
Move 80: move disk 4 from 3 to 4
Move 81: move disk 4 from 4 to 5
Move 82: move disk 6 from S to 1
Move 83: move disk 6 from 1 to 2
Move 84: move disk 6 from 2 to 3
Move 85: move disk 6 from 3 to 4
Move 86: move disk 1 from 5 to 1
Move 87: move disk 1 from 1 to 2
Move 88: move disk 1 from 2 to 3
Move 89: move disk 2 from 5 to 1
Move 90: move disk 1 from 3 to 4
Move 91: move disk 1 from 4 to 5
Move 92: move disk 2 from 1 to 2
Move 93: move disk 2 from 2 to 3
Move 94: move disk 2 from 3 to 4
Move 95: move disk 3 from 5 to 1
Move 96: move disk 2 from 3 to 4
Move 97: move disk 2 from 4 to 5
Move 98: move disk 3 from 1 to 2
Move 99: move disk 3 from 2 to 3
Move 100: move disk 3 from 3 to 4
... (Could not get to work so this was done by hand)
Move 346: move disk 2 from 3 to 4
Move 347: move disk 2 from 4 to 5
Move 348: move disk 3 from 1 to 2
Move 349: move disk 3 from 2 to 3
Move 350: move disk 3 from 3 to 4
Move 351: move disk 4 from 5 to 1
Move 352: move disk 3 from 3 to 4
Move 353: move disk 3 from 4 to 5
Move 354: move disk 4 from 1 to 2
Move 355: move disk 4 from 2 to 3
Move 356: move disk 4 from 3 to 4
Move 357: move disk 5 from 5 to 1
```

Move 358: move disk 4 from 3 to 4
Move 359: move disk 4 from 4 to 5
Move 360: move disk 5 from 1 to 2
Move 361: move disk 5 from 2 to 3
Move 362: move disk 5 from 3 to 4
Move 363: move disk 6 from 5 to 1
Move 364: move disk 5 from 3 to 4
Move 365: move disk 5 from 4 to 5
Move 366: move disk 6 from 1 to 2
Move 367: move disk 6 from 2 to 3
Move 368: move disk 6 from 3 to 4
Move 369: move disk 1 from 5 to 1
Move 370: move disk 1 from 1 to 2
Move 371: move disk 1 from 2 to 3
Move 372: move disk 2 from 5 to 1
Move 373: move disk 1 from 3 to 4
Move 374: move disk 1 from 4 to 5
Move 375: move disk 2 from 1 to 2
Move 376: move disk 2 from 2 to 3
Move 377: move disk 2 from 3 to 4
Move 378: move disk 3 from 5 to 1
Move 379: move disk 2 from 3 to 4
Move 380: move disk 2 from 4 to 5
Move 381: move disk 3 from 1 to 2
Move 382: move disk 3 from 2 to 3
Move 383: move disk 3 from 3 to 4
Move 384: move disk 4 from 5 to 1
Move 385: move disk 3 from 3 to 4
Move 386: move disk 3 from 4 to 5
Move 387: move disk 4 from 1 to 2
Move 388: move disk 4 from 2 to 3
Move 389: move disk 4 from 3 to 4
Move 390: move disk 5 from 5 to 1
Move 391: move disk 4 from 3 to 4
Move 392: move disk 4 from 4 to 5
Move 393: move disk 5 from 1 to 2
Move 394: move disk 5 from 2 to 3
Move 395: move disk 5 from 3 to 4
Move 396: move disk 1 from 5 to 1
Move 397: move disk 1 from 1 to 2
Move 398: move disk 1 from 2 to 3
Move 399: move disk 2 from 5 to 1
Move 400: move disk 1 from 3 to 4
Move 401: move disk 1 from 4 to 5
Move 402: move disk 2 from 1 to 2

```
Move 403: move disk 2 from 2 to 3
Move 404: move disk 2 from 3 to 4
Move 405: move disk 3 from 5 to 1
Move 406: move disk 2 from 3 to 4
Move 407: move disk 2 from 4 to 5
Move 408: move disk 3 from 1 to 2
Move 409: move disk 3 from 2 to 3
Move 410: move disk 3 from 3 to 4
Move 411: move disk 4 from 5 to 1
Move 412: move disk 3 from 3 to 4
Move 413: move disk 3 from 4 to 5
Move 414: move disk 4 from 1 to 2
Move 415: move disk 4 from 2 to 3
Move 416: move disk 4 from 3 to 4
Move 417: move disk 1 from 5 to 1
Move 418: move disk 1 from 1 to 2
Move 419: move disk 1 from 2 to 3
Move 420: move disk 2 from 5 to 1
Move 421: move disk 1 from 3 to 4
Move 422: move disk 1 from 4 to 5
Move 423: move disk 2 from 1 to 2
Move 424: move disk 2 from 2 to 3
Move 425: move disk 2 from 3 to 4
Move 426: move disk 3 from 5 to 1
Move 427: move disk 2 from 3 to 4
Move 428: move disk 2 from 4 to 5
Move 429: move disk 3 from 1 to 2
Move 430: move disk 3 from 2 to 3
Move 431: move disk 3 from 3 to 4
Move 432: move disk 1 from 5 to 1
Move 433: move disk 1 from 1 to 2
Move 434: move disk 1 from 2 to 3
Move 435: move disk 2 from 5 to 1
Move 436: move disk 1 from 3 to 4
Move 437: move disk 1 from 4 to 5
Move 438: move disk 2 from 1 to 2
Move 439: move disk 2 from 2 to 3
Move 440: move disk 2 from 3 to 4
Move 441: move disk 2 from 4 to D
Move 442: move disk 1 from 5 to 1
Move 443: move disk 1 from 1 to 2
Move 444: move disk 1 from 2 to 3
Move 445: move disk 1 from 3 to 4
Move 446: move disk 1 from 4 to D
Total moves for the disk was: 446
```

```
Please enter how many disks we will be playing with today:
10
Move 1: move disk 1 from S to 1
Move 2: move disk 1 from 1 to 2
Move 3: move disk 1 from 2 to 3
Move 4: move disk 1 from 3 to 4
Move 5: move disk 1 from 4 to 5
Move 6: move disk 2 from S to 1
Move 7: move disk 2 from 1 to 2
Move 8: move disk 2 from 2 to 3
Move 9: move disk 2 from 3 to 4
Move 10: move disk 1 from 5 to 1
Move 11: move disk 1 from 1 to 2
Move 12: move disk 1 from 2 to 3
Move 13: move disk 2 from 4 to 5
Move 14: move disk 1 from 3 to 4
Move 15: move disk 1 from 4 to 5
Move 16: move disk 3 from S to 1
Move 17: move disk 3 from 1 to 2
Move 18: move disk 3 from 2 to 3
Move 19: move disk 3 from 3 to 4
Move 20: move disk 1 from 5 to 1
Move 21: move disk 1 from 1 to 2
Move 22: move disk 1 from 2 to 3
Move 23: move disk 2 from 5 to 1
Move 24: move disk 1 from 3 to 4
Move 25: move disk 1 from 4 to 5
Move 26: move disk 2 from 1 to 2
Move 27: move disk 2 from 2 to 3
Move 28: move disk 2 from 3 to 4
Move 29: move disk 3 from 4 to 5
Move 30: move disk 2 from 3 to 4
Move 31: move disk 2 from 4 to 5
Move 32: move disk 4 from S to 1
Move 33: move disk 4 from 1 to 2
Move 34: move disk 4 from 2 to 3
Move 35: move disk 4 from 3 to 4
Move 36: move disk 1 from 5 to 1
Move 37: move disk 1 from 1 to 2
Move 38: move disk 1 from 2 to 3
Move 39: move disk 2 from 5 to 1
Move 40: move disk 1 from 3 to 4
Move 41: move disk 1 from 4 to 5
Move 42: move disk 2 from 1 to 2
```

```
Move 43: move disk 2 from 2 to 3
Move 44: move disk 2 from 3 to 4
Move 45: move disk 3 from 5 to 1
Move 46: move disk 2 from 3 to 4
Move 47: move disk 2 from 4 to 5
Move 48: move disk 3 from 1 to 2
Move 49: move disk 3 from 2 to 3
Move 50: move disk 3 from 3 to 4
Move 51: move disk 4 from 4 to 5
Move 52: move disk 3 from 3 to 4
Move 53: move disk 3 from 4 to 5
Move 54: move disk 5 from S to 1
Move 55: move disk 5 from 1 to 2
Move 56: move disk 5 from 2 to 3
Move 57: move disk 5 from 3 to 4
Move 58: move disk 1 from 5 to 1
Move 59: move disk 1 from 1 to 2
Move 60: move disk 1 from 2 to 3
Move 61: move disk 2 from 5 to 1
Move 62: move disk 1 from 3 to 4
Move 63: move disk 1 from 4 to 5
Move 64: move disk 2 from 1 to 2
Move 65: move disk 2 from 2 to 3
Move 66: move disk 2 from 3 to 4
Move 67: move disk 3 from 5 to 1
Move 68: move disk 2 from 3 to 4
Move 69: move disk 2 from 4 to 5
Move 70: move disk 3 from 1 to 2
Move 71: move disk 3 from 2 to 3
Move 72: move disk 3 from 3 to 4
Move 73: move disk 4 from 5 to 1
Move 74: move disk 3 from 3 to 4
Move 75: move disk 3 from 4 to 5
Move 76: move disk 4 from 1 to 2
Move 77: move disk 4 from 2 to 3
Move 78: move disk 4 from 3 to 4
Move 79: move disk 5 from 4 to 5
Move 80: move disk 4 from 3 to 4
Move 81: move disk 4 from 4 to 5
Move 82: move disk 6 from S to 1
Move 83: move disk 6 from 1 to 2
Move 84: move disk 6 from 2 to 3
Move 85: move disk 6 from 3 to 4
Move 86: move disk 1 from 5 to 1
Move 87: move disk 1 from 1 to 2
```

```
Move 88: move disk 1 from 2 to 3
Move 89: move disk 2 from 5 to 1
Move 90: move disk 1 from 3 to 4
Move 91: move disk 1 from 4 to 5
Move 92: move disk 2 from 1 to 2
Move 93: move disk 2 from 2 to 3
Move 94: move disk 2 from 3 to 4
Move 95: move disk 3 from 5 to 1
Move 96: move disk 2 from 3 to 4
Move 97: move disk 2 from 4 to 5
Move 98: move disk 3 from 1 to 2
Move 99: move disk 3 from 2 to 3
Move 100: move disk 3 from 3 to 4
... (Could not get to work so this was done by hand)
Move 455: move disk 2 from 3 to 4
Move 456: move disk 2 from 4 to 5
Move 457: move disk 3 from 1 to 2
Move 458: move disk 3 from 2 to 3
Move 459: move disk 3 from 3 to 4
Move 460: move disk 4 from 5 to 1
Move 461: move disk 3 from 3 to 4
Move 462: move disk 3 from 4 to 5
Move 463: move disk 4 from 1 to 2
Move 464: move disk 4 from 2 to 3
Move 465: move disk 4 from 3 to 4
Move 466: move disk 5 from 5 to 1
Move 467: move disk 4 from 3 to 4
Move 468: move disk 4 from 4 to 5
Move 469: move disk 5 from 1 to 2
Move 470: move disk 5 from 2 to 3
Move 471: move disk 5 from 3 to 4
Move 472: move disk 6 from 5 to 1
Move 473: move disk 5 from 3 to 4
Move 474: move disk 5 from 4 to 5
Move 475: move disk 6 from 1 to 2
Move 476: move disk 6 from 2 to 3
Move 477: move disk 6 from 3 to 4
Move 478: move disk 1 from 5 to 1
Move 479: move disk 1 from 1 to 2
Move 480: move disk 1 from 2 to 3
Move 481: move disk 2 from 5 to 1
Move 482: move disk 1 from 3 to 4
Move 483: move disk 1 from 4 to 5
Move 484: move disk 2 from 1 to 2
Move 485: move disk 2 from 2 to 3
```

Move 486: move disk 2 from 3 to 4
Move 487: move disk 3 from 5 to 1
Move 488: move disk 2 from 3 to 4
Move 489: move disk 2 from 4 to 5
Move 490: move disk 3 from 1 to 2
Move 491: move disk 3 from 2 to 3
Move 492: move disk 3 from 3 to 4
Move 493: move disk 4 from 5 to 1
Move 494: move disk 3 from 3 to 4
Move 495: move disk 3 from 4 to 5
Move 496: move disk 4 from 1 to 2
Move 497: move disk 4 from 2 to 3
Move 498: move disk 4 from 3 to 4
Move 499: move disk 5 from 5 to 1
Move 500: move disk 4 from 3 to 4
Move 501: move disk 4 from 4 to 5
Move 502: move disk 5 from 1 to 2
Move 503: move disk 5 from 2 to 3
Move 504: move disk 5 from 3 to 4
Move 505: move disk 1 from 5 to 1
Move 506: move disk 1 from 1 to 2
Move 507: move disk 1 from 2 to 3
Move 508: move disk 2 from 5 to 1
Move 509: move disk 1 from 3 to 4
Move 510: move disk 1 from 4 to 5
Move 511: move disk 2 from 1 to 2
Move 512: move disk 2 from 2 to 3
Move 513: move disk 2 from 3 to 4
Move 514: move disk 3 from 5 to 1
Move 515: move disk 2 from 3 to 4
Move 516: move disk 2 from 4 to 5
Move 517: move disk 3 from 1 to 2
Move 518: move disk 3 from 2 to 3
Move 519: move disk 3 from 3 to 4
Move 520: move disk 4 from 5 to 1
Move 521: move disk 3 from 3 to 4
Move 522: move disk 3 from 4 to 5
Move 523: move disk 4 from 1 to 2
Move 524: move disk 4 from 2 to 3
Move 525: move disk 4 from 3 to 4
Move 526: move disk 1 from 5 to 1
Move 527: move disk 1 from 1 to 2
Move 528: move disk 1 from 2 to 3
Move 529: move disk 2 from 5 to 1
Move 530: move disk 1 from 3 to 4

```
Move 531: move disk 1 from 4 to 5
Move 532: move disk 2 from 1 to 2
Move 533: move disk 2 from 2 to 3
Move 534: move disk 2 from 3 to 4
Move 535: move disk 3 from 5 to 1
Move 536: move disk 2 from 3 to 4
Move 537: move disk 2 from 4 to 5
Move 538: move disk 3 from 1 to 2
Move 539: move disk 3 from 2 to 3
Move 540: move disk 3 from 3 to 4
Move 541: move disk 1 from 5 to 1
Move 542: move disk 1 from 1 to 2
Move 543: move disk 1 from 2 to 3
Move 544: move disk 2 from 5 to 1
Move 545: move disk 1 from 3 to 4
Move 546: move disk 1 from 4 to 5
Move 547: move disk 2 from 1 to 2
Move 548: move disk 2 from 2 to 3
Move 549: move disk 2 from 3 to 4
Move 550: move disk 2 from 4 to D
Move 551: move disk 1 from 5 to 1
Move 552: move disk 1 from 1 to 2
Move 553: move disk 1 from 2 to 3
Move 554: move disk 1 from 3 to 4
Move 555: move disk 1 from 4 to D
Total moves for the disk was: 555
*/
```

PROGRAM 2

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <chrono>

const int PAGE_SIZE = 2000;
const int ACTIVE_MEMORY_SIZE = 1000;

int calculatePageNumber(int i, int j) {
    return (i * j) * 10;
}

std::pair<int, int> calculatePageTransfersRowMajor() {
```

```
    int totalReads = 0;
    int totalWrites = 0;
    std::unordered_map<int, bool> pageTable;
    std::list<int> activePages;

    for (int i = 1; i <= 4000; ++i) { // Rows
        for (int j = 1; j <= 4000; ++j) { // Columns
            // Access A[i, j]
            int page = calculatePageNumber(i, j);
            if (pageTable.find(page) == pageTable.end()) {
                if (activePages.size() < ACTIVE_MEMORY_SIZE) {
                    activePages.push_back(page);
                } else {
                    int evictedPage = activePages.front();
                    activePages.pop_front();
                    activePages.push_back(page);
                    pageTable.erase(evictedPage);
                    ++totalWrites;
                }
                pageTable[page] = true;
                ++totalReads;
            }

            // Access B[i, j]
            page = calculatePageNumber(i, j);
            if (pageTable.find(page) == pageTable.end()) {
                if (activePages.size() < ACTIVE_MEMORY_SIZE) {
                    activePages.push_back(page);
                } else {
                    int evictedPage = activePages.front();
                    activePages.pop_front();
                    activePages.push_back(page);
                    pageTable.erase(evictedPage);
                    ++totalWrites;
                }
                pageTable[page] = true;
                ++totalReads;
            }
        }
    }

    return std::make_pair(totalReads, totalWrites);
}

int main() {
```

```
    std::chrono::steady_clock::time_point startTime =
std::chrono::steady_clock::now();

    std::pair<int, int> pageTransfers = calculatePageTransfersRowMajor();
    int totalReads = pageTransfers.first;
    int totalWrites = pageTransfers.second;

    std::chrono::steady_clock::time_point endTime =
std::chrono::steady_clock::now();
    std::chrono::duration<double> duration =
std::chrono::duration_cast<std::chrono::duration<double>>(endTime - startTime);

    std::cout << "Total reads: " << totalReads << std::endl;
    std::cout << "Total writes: " << totalWrites << std::endl;
    std::cout << "Execution time: " << duration.count() << " seconds" <<
std::endl;

    return 0;
}

//g++ Assignment2.cpp -o Assignment && ./Assignment to build and run

/*
DATA STRUCTURES USED - Map, doubly linked list
Hypothesis - I believe Row Major Order will be shorter in length of time than if
it were in column major order.
Conclusion - They're both the same length of time. I was wrong.

The code uses an unordered map pageTable to track the pages currently in main
memory. It maps page numbers to boolean
values to indicate whether a page is present in memory.
A list activePages is used to simulate the LRU replacement strategy. It keeps
track of the most recently used pages,
with the least recently used page being at the front of the list.
In this row-major order version, the outer loop iterates over the rows (i), and
the inner loop iterates over the
columns (j). This ensures that elements from the same row are accessed
consecutively.
Both row-major and column-major order versions access both arrays A and B in the
same order. The difference lies in
the order of iterating over the dimensions (i and j) and the arguments passed to
the calculatePageNumber function.

Time Complexity -
```

The time complexity is O(N^2) where n is the max cases which is i = 4000 and j = 4000. So it is O(n^2) where n = 4000.
Since we're using the hash table, the active memory size is the n when measure the O(n), so the case for Space complexity
is O(1).
Space Complexity -
pageTable: An unordered map that can hold a maximum of 1000 pages in memory. As the maximum active memory size is defined
as 1000, the space complexity of the pageTable is O(ACTIVE_MEMORY_SIZE), which is O(1) in this case.
activePages: A list that simulates the LRU replacement strategy. It can hold a maximum of 1000 page numbers. Thus, the
space complexity of the activePages list is O(ACTIVE_MEMORY_SIZE), which is O(1) as well.

```
//Read 1
Total reads: 16000000
Total writes: 15999000
Execution time: 13.311 seconds

//Read 2
Total reads: 16000000
Total writes: 15999000
Execution time: 13.3082 seconds

//Read 3
Total reads: 16000000
Total writes: 15999000
Execution time: 13.445 seconds
*/
```

PROGRAM 2

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <chrono>

const int PAGE_SIZE = 2000;
const int ACTIVE_MEMORY_SIZE = 1000;

int calculatePageNumber(int i, int j) {
    return (i * j) * 10;
```

```
}

std::pair<int, int> calculatePageTransfersColumnMajor() {
    int totalReads = 0;
    int totalWrites = 0;
    std::unordered_map<int, bool> pageTable;
    std::list<int> activePages;

    for (int j = 1; j <= 4000; ++j) { // Columns
        for (int i = 1; i <= 4000; ++i) { // Rows
            // Access A[i, j]
            int page = calculatePageNumber(i, j);
            if (pageTable.find(page) == pageTable.end()) {
                if (activePages.size() < ACTIVE_MEMORY_SIZE) {
                    activePages.push_back(page);
                } else {
                    int evictedPage = activePages.front();
                    activePages.pop_front();
                    activePages.push_back(page);
                    pageTable.erase(evictedPage);
                    ++totalWrites;
                }
                pageTable[page] = true;
                ++totalReads;
            }

            // Access B[i, j]
            page = calculatePageNumber(i, j);
            if (pageTable.find(page) == pageTable.end()) {
                if (activePages.size() < ACTIVE_MEMORY_SIZE) {
                    activePages.push_back(page);
                } else {
                    int evictedPage = activePages.front();
                    activePages.pop_front();
                    activePages.push_back(page);
                    pageTable.erase(evictedPage);
                    ++totalWrites;
                }
                pageTable[page] = true;
                ++totalReads;
            }
        }
    }

    return std::make_pair(totalReads, totalWrites);
```

```
}

int main() {
    std::chrono::steady_clock::time_point startTime =
std::chrono::steady_clock::now();

    std::pair<int, int> pageTransfers = calculatePageTransfersColumnMajor();
    int totalReads = pageTransfers.first;
    int totalWrites = pageTransfers.second;

    std::chrono::steady_clock::time_point endTime =
std::chrono::steady_clock::now();
    std::chrono::duration<double> duration =
std::chrono::duration_cast<std::chrono::duration<double>>(endTime - startTime);

    std::cout << "Total reads: " << totalReads << std::endl;
    std::cout << "Total writes: " << totalWrites << std::endl;
    std::cout << "Execution time: " << duration.count() << " seconds" <<
std::endl;

    return 0;
}

//g++ Assignment2.cpp -o Assignment && ./Assignment to build and run

/*
//
HYPOTHESIS - I believe the Row Major Order will be faster given in class it's
described as being faster.
Conclusion - They are both essentially the same when being conducted.

DATA STRUCTURES USED - Map, doubly linked list

The code uses an unordered map pageTable to track the pages currently in main
memory. It maps page numbers to boolean
values to indicate whether a page is present in memory.
A list activePages is used to simulate the LRU replacement strategy. It keeps
track of the most recently used pages,
with the least recently used page being at the front of the list.
In this column-major order version, the outer loop iterates over the columns (j),
and the inner loop iterates over the
rows (i). This ensures that elements from the same column are accessed
consecutively.
Both row-major and column-major order versions access both arrays A and B in the
same order. The difference lies in
```

the order of iterating over the dimensions (i and j) and the arguments passed to the calculatePageNumber function.

//CONCLUSION - The Row Major order and Column Major order run the same time.

Time Complexity -
The time complexity is O(N^2) where n is the max cases which is i = 4000 and j = 4000. So it is O(n^2) where n = 4000.
Since we're using the hash table, the active memory size is the n when measure the O(n), so the case for Space complexity
is O(1).
Space Complexity -
pageTable: An unordered map that can hold a maximum of 1000 pages in memory. As the maximum active memory size is defined
as 1000, the space complexity of the pageTable is O(ACTIVE_MEMORY_SIZE), which is O(1) in this case.
activePages: A list that simulates the LRU replacement strategy. It can hold a maximum of 1000 page numbers. Thus, the
space complexity of the activePages list is O(ACTIVE_MEMORY_SIZE), which is O(1) as well.

//Read 1
Total reads: 16000000
Total writes: 15999000
Execution time: 13.3445 seconds

//Read 2
Total reads: 16000000
Total writes: 15999000
Execution time: 13.399 seconds

//Read 3
Total reads: 16000000
Total writes: 15999000
Execution time: 13.368 seconds
*/

PROGRAM 3

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <chrono>
```

```
#include <random>

using namespace std;
using namespace chrono;

//now for the sorting to actually begin
int partition(vector<int>& arr, int start, int end) {
    int pivot = arr[start]; //pivot = x
    int i = start + 1;
    int j = end;

    while (true) {
        while (i <= j && arr[i] <= pivot)
            i++;

        while (i <= j && arr[j] > pivot)
            j--;

        if (i > j)
            break;

        swap(arr[i], arr[j]);
    }

    swap(arr[start], arr[j]);
    return j;
}

void quickSort(vector<int>& arr, int start, int end) {
    if (start < end) {
        int pivotIndex = partition(arr, start, end);
        quickSort(arr, start, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, end);
    }
}

int main() {
    int n = 1000;
    vector<int> arr(n);

    // Generate random numbers for the array
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dis(1, 1000);
    for (int i = 0; i < n; i++) {
```

```
        arr[i] = dis(gen);
    }

    //sort(arr.begin(), arr.end()); // Sort the array in ascending order to make
the worst case which is O(n^2)
                                  // Delete to have average case of O(n*logn)
    time_point<steady_clock> start, end;
    start = steady_clock::now();
    quickSort(arr, 0, n-1);
    end = steady_clock::now();
    duration<double, nano> elapsed_time = end - start;

    cout << "Time: " << elapsed_time.count() << " nanoseconds" << endl;

    return 0;
}
/*use
g++ Assignment3.cpp -o Assignment && ./Assignment
to run the code */
/*
Hypothesis - The sort algorithm will have an extremely fast time and space
complexity,
however I want to see how it works given that it is described as a less than
reputable
sorting algorithm by Professor Leiss.

Conclusion - It is a fast algorithm only when it is not previously sorted. If the
numbers
are random and they are already sorted. Then the algorithm quickly turns into an
extremely
slow moving sorting algorithm.

DATA STRUCTURES USED - Vectors, Number engine generator, uniiform distribution
oject

The provided code implements the QuickSort algorithm to sort a vector of
integers.
It uses the partition function to choose a pivot element and rearrange the
elements
in the vector such that all elements smaller than the pivot are placed before it
and
all elements greater than the pivot are placed after it. The quickSort function
recursively calls itself to sort the subarrays on the left and right of the
pivot.
```

When the input array is already sorted (either in ascending or descending order) or
contains many elements with the same value.
In such cases, if the pivot selection and partitioning strategy are not optimized,
the partitioning step can create highly unbalanced partitions. If the pivot consistently selects
the smallest or largest element, the partitioning will not divide the array evenly, leading to one
partition significantly larger than the other.

Time Complexity -
As a result, the recursion depth becomes equal to the size of the input array, and each recursive
call processes a reduced portion of the input array by only one element at a time. This scenario
leads to an overall time complexity of O(n^2), as each element needs to be compared and moved
multiple times.

It's important to note that the code I provided does not include any specific optimizations for
pivot selection or partitioning strategy, which can contribute to the worst-case time complexity.
However, in practice, randomized pivot selection or other techniques can be employed to mitigate
the chances of encountering the worst-case scenario and achieve better average-case or expected
time complexity of O(n log n).

Space Complexity -
The space complexity of the given code is O(log n) in the worst case, where n is the number of
elements in the array. This space complexity arises from the recursive calls made in the quickSort
function.

In the quickSort function, the recursive calls are made for two subproblems, each with approximately
half the size of the original problem. This means that in the worst case, the maximum depth of
the recursion would be log n.

At each level of the recursion, a constant amount of additional space is used for the variables

```
pivotIndex, start, and end. Therefore, the space required for each level of
recursion is constant.
Since the maximum depth of the recursion is log n, the overall space complexity
is O(log n).

//O(n log n)
//Timing 1
Time: 0 nanoseconds
//Timing 2
Time: 1.003e+006 nanoseconds
//Timing 3
Time: 0 nanoseconds

//O(n^2) //uncomment the sort mechanic
//Timing 1
Time: 1.001e+006 nanoseconds
//Timing 2
Time: 979000 nanoseconds
//Timing 3
Time: 1.004e+006 nanoseconds
*/
```

PROGRAM 4

```
import numpy as np
import time

# Function to measure the time taken for matrix addition

"""
    Perform matrix addition of two n x n matrices and measure the execution time.

    Parameters:
    - n: The size of the matrices.
    - version: The version of the matrix addition algorithm to use (1 or 2).

    Returns:
    - The execution time in seconds.
"""

def matrix_addition(n, version):
    A = np.random.rand(n, n)  # Initialize matrix A with random values
    B = np.random.rand(n, n)  # Initialize matrix B with random values
    C = np.zeros((n, n))      # Initialize matrix C with zeros
```

```
    start_time = time.time()

    if version == 1:
        for i in range(n):
            for j in range(n):
                C[i, j] = A[i, j] + B[i, j]
    elif version == 2:
        for j in range(n):
            for i in range(n):
                C[i, j] = A[i, j] + B[i, j]

    end_time = time.time()
    return end_time - start_time

# Sequence of values for n
sequence = [128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]

# Measure timings for both versions
for n in sequence:
    time_version_1 = matrix_addition(n, 1)
    time_version_2 = matrix_addition(n, 2)
    print(f"n = {n}, Version 1 time = {time_version_1}, Version 2 time =
{time_version_2}")

#run using
#python -u Assignment4.py
#python3 -u Assignment4.py - wsl

#Timing
"""
Hypothesis - The switching of the cases given whether it measures I or J first
won't matter.
It will still end up as the same measured timings.

Conclusion - Measuring the version where i is the first for loop went almost
twice as fast as
when using the J loop for. I was to believe that a simple switch of the algorithm
wouldn't change
anything and that it would simply be just as fast.

The code measures the execution time of two versions of matrix addition algorithm
for different
matrix sizes. It generates random matrices A and B of size n x n, initializes
matrix C with zeros,
```

and then performs the matrix addition using either version 1 or version 2. The execution time is
measured using the time.time() function before and after the matrix addition, and the difference
is calculated.

The matrix_addition function takes two parameters: n for the matrix size and version to determine
which algorithm version to use. It returns the execution time in seconds.

The sequence list defines the values of n for which the timings will be measured. The code then
iterates over this sequence, calls the matrix_addition function for both versions, and prints the results.

Assignment 4 Notes

Note - THAT EVERYTHING IS IN SECONDS

n = 128, Version 1 time = 0.00435638427734375, Version 2 time = 0.0042002201080322266
n = 256, Version 1 time = 0.01934814453125, Version 2 time = 0.01830458641052246
n = 512, Version 1 time = 0.07316088676452637, Version 2 time = 0.07638120651245117
n = 1024, Version 1 time = 0.29168248176574707, Version 2 time = 0.40419745445251465
n = 2048, Version 1 time = 1.1686546802520752, Version 2 time = 1.7977614402770996
n = 4096, Version 1 time = 4.697006464004517, Version 2 time = 7.052334308624268
n = 8192, Version 1 time = 18.970118761062622, Version 2 time = 28.37813973426819
n = 16384, Version 1 time = 75.34112977981567, Version 2 time = 116.23827624320984
n = 32768, Version 1 time = 306.2558307647705, Version 2 time = 657.3897912502289

Run 2
n = 128, Version 1 time = 0.0044901371002197266, Version 2 time = 0.004387855529785156
n = 256, Version 1 time = 0.01791524887084961, Version 2 time = 0.018084049224853516
n = 512, Version 1 time = 0.07451748847961426, Version 2 time = 0.08118438720703125
n = 1024, Version 1 time = 0.2920260429382324, Version 2 time = 0.41611385345458984
n = 2048, Version 1 time = 1.1737470626831055, Version 2 time = 1.7096195220947266

n = 4096, Version 1 time = 4.789830923080444, Version 2 time = 6.918642997741699
n = 8192, Version 1 time = 18.919447422027588, Version 2 time = 27.63094449043274
n = 16384, Version 1 time = 76.12821388244629, Version 2 time =
111.71175122261047
n = 32768, Version 1 time = 304.8916037082672, Version 2 time = 547.7969162464142

Run 3
n = 128, Version 1 time = 0.004434823989868164, Version 2 time =
0.004378080368041992
n = 256, Version 1 time = 0.01803445816040039, Version 2 time =
0.018113374710083008
n = 512, Version 1 time = 0.07565569877624512, Version 2 time =
0.0820000171661377
n = 1024, Version 1 time = 0.293764591217041, Version 2 time = 0.4276130199432373
n = 2048, Version 1 time = 1.1761465072631836, Version 2 time =
1.7011590003967285
n = 4096, Version 1 time = 4.734057426452637, Version 2 time = 6.901985168457031
n = 8192, Version 1 time = 19.026060581207275, Version 2 time =
27.722479343414307
n = 16384, Version 1 time = 76.27064728736877, Version 2 time =
131.99759316444397
n = 32768, Version 1 time = 306.81384348869324, Version 2 time =
712.4329967498779

Matrix Initialization: The code initializes three matrices A, B, and C with size
n x n. Initializing
each element of the matrices has a time complexity of O(1). Therefore, the time
complexity of matrix
initialization is O(n^2).
Matrix Addition: The code performs matrix addition using two nested loops. For
each element in the matrices
A and B, it performs a single addition operation. Since there are n x n elements
in total, the time complexity
of matrix addition is O(n^2).
Timing Measurements: The code measures the execution time of each matrix addition
operation using the time.time()
function. The time complexity of timing measurements is negligible compared to
the matrix addition operations.
Iterating Over Sequence: The code iterates over the sequence of values for n.
Since the sequence has a fixed
length, the time complexity of this part is O(1).
Overall, the time complexity of the code can be expressed as O(n^2), where n is
the size of the matrices.
As for the space complexity, the code initializes three matrices A, B, and C,
each with size n x n. Therefore,

```
the space complexity is O(n^2) to store the matrices.
Note that the numpy library is used for matrix operations, which internally
utilizes optimized routines and
data structures. The space complexity may also depend on the internal memory
management of numpy, but it is
still dominated by the size of the matrices, which is O(n^2).
"""
```

PROGRAM 5

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

int main() {
    struct rlimit mem_limit;
    getrlimit(RLIMIT_AS, &mem_limit);
    long long available_memory = mem_limit.rlim_cur / 3;  // Divide by 3 for each
array
    int** AR1 = NULL;
    int** AR2 = NULL;
    int** AR3 = NULL;
    int** AR4 = NULL;
    int* newElementAR1 = NULL;
    int* newElementAR2 = NULL;
    int* newElementAR3 = NULL;

    int element_size = 1024 * 1024;  // 1MB
    struct timeval start, end;
    long long elapsed_micros, elapsed_seconds, elapsed_milliseconds;

    //allocates the memory by 3m, m = count or size
    gettimeofday(&start, NULL);
    int count = 0;
    while (count < 130001) { // 130000*3 = m = 390,000
      if (available_memory < element_size) break;
        int* Array1 = (int*)malloc(element_size);
      if(Array1 == NULL) break;
        int* Array2 = (int*)malloc(element_size);
      if(Array2 == NULL) break;
        int* Array3 = (int*)malloc(element_size);
      if(Array3 == NULL) break;
        AR1 = (int**)realloc(AR1, (count + 1) * sizeof(int*));
        AR2 = (int**)realloc(AR2, (count + 1) * sizeof(int*));
```

```
        AR3 = (int**)realloc(AR3, (count + 1) * sizeof(int*));
        AR1[count] = Array1;
        AR2[count] = Array2;
        AR3[count] = Array3;
        count++;
        //printf("Number of elements in AR1, AR2, AR3: %d\n", count);
        available_memory -= element_size;
    }
    gettimeofday(&end, NULL);
    elapsed_micros = (end.tv_sec - start.tv_sec) * 1000000LL + (end.tv_usec -
start.tv_usec); //microseconds
    elapsed_seconds = elapsed_micros / 1000000; // seconds
    printf("Time taken for initial allocation: %lld seconds\n", elapsed_seconds);
    printf("Number of elements in AR1, AR2, AR3: %d\n", count);
    gettimeofday(&start, NULL);
    //frees every array of i % 2 = 0
    for (int i = 0; i < count; i += 2) {
        free(AR1[i]);
        free(AR2[i]);
        free(AR3[i]);
    }
    gettimeofday(&end, NULL);
    elapsed_micros = (end.tv_sec - start.tv_sec) * 1000000LL + (end.tv_usec -
start.tv_usec);
    elapsed_milliseconds = elapsed_micros / 1000; // milliseconds
    printf("Time taken for deallocation of even number elements: %lld
milliseconds\n", elapsed_milliseconds);

    //now to reallocate AR4 as we've only deallocated memory from AR1, AR2, AR3
of 2. We cannot reallocate
    gettimeofday(&start, NULL);
    int** tempAR4 = NULL;
    int ar4_count = 0;
    int ar4_element_size = 1024 * 1024 * 1.25;  // 1.25MB
    while (ar4_count < 13000) {
        if (available_memory < ar4_element_size)
            break;
        int* newElementAR4 = (int*)malloc(ar4_element_size);
        if (newElementAR4 == NULL)
            break;
        tempAR4 = (int**)realloc(tempAR4, (ar4_count + 1) * sizeof(int*));
        tempAR4[ar4_count] = newElementAR4;
        ar4_count++;
        available_memory -= ar4_element_size;
        //printf("Number of elements in AR4: %d\n", ar4_count);
```

```
    }
    AR4 = tempAR4;
    gettimeofday(&end, NULL);
    elapsed_micros = (end.tv_sec - start.tv_sec) * 1000000LL + (end.tv_usec -
start.tv_usec);
    elapsed_seconds = elapsed_micros / 1000000; // seconds
    printf("Time taken for allocation to AR4: %lld seconds\n", elapsed_seconds);
    printf("Number of elements in AR4: %d\n", ar4_count);

    for (int i = count-2; i >= 0; i-=2) { //now frees odd memory
        free(AR1[i]);
        free(AR2[i]);
        free(AR3[i]);
    }
    free(AR1);
    free(AR2);
    free(AR3);

    for (int i = 0; i < ar4_count; i++) { //frees all of AR4 memory
        free(AR4[i]);
    }
    free(AR4);

    return 0;
}

/*
DATA STRUCTURES USED - Pointers, Arrays

The code measures the time taken for memory allocation and deallocation
operations using dynamic memory allocation
(malloc) and reallocation (realloc) in C.

The getrlimit function retrieves the current memory limit, and the available
memory is calculated by dividing it by
3 to allocate memory for three arrays: AR1, AR2, and AR3.

The code uses malloc to allocate memory blocks of size element_size (1MB) for
Array1, Array2, and Array3. It then
uses realloc to resize the arrays AR1, AR2, and AR3 to hold the newly allocated
memory blocks. This process is
repeated until either the available memory is insufficient or the count reaches
130,001. (130,001 was used for AR1,
AR2, and AR3 because it's what my machine on my computer at home can handle).
```

After the initial allocation, the code measures the time taken for deallocation by freeing memory for every
even-indexed element in AR1, AR2, and AR3.

Next, the code allocates memory for AR4 using malloc for an element size of ar4_element_size (1.25MB). It keeps track
of the count and the available memory. This process is repeated until either the available memory is insufficient
or the count reaches 13,000. (13,000 being used for AR4 is what my machine on my computer at home can handle).

Finally, the code frees the odd-indexed memory in AR1, AR2, and AR3 and frees all the memory blocks in AR4. It also
frees the arrays AR1, AR2, AR3, and AR4.

Time Complexity -
Initial Allocation: The while loop iterates until the count reaches 130,001 or the available memory is insufficient.
Inside the loop, memory allocation (malloc) and reallocation (realloc) operations are performed. Both malloc and
realloc have a time complexity of O(1) on average. Therefore, the time complexity of this section is O(n), where n
is the number of iterations in the loop.
Deallocation of Even-Indexed Elements: The for loop iterates over the even-indexed elements of AR1, AR2, and AR3
and performs memory deallocation (free). The time complexity of the deallocation operation (free) is O(1). Therefore,
the time complexity of this section is O(m), where m is the number of even-indexed elements.
Allocation to AR4: The while loop iterates until the ar4_count reaches 13,000 or the available memory is insufficient.
Inside the loop, memory allocation (malloc) and reallocation (realloc) operations are performed. Again, both malloc and
realloc have a time complexity of O(1) on average. Therefore, the time complexity of this section is O(k), where k is
the number of iterations in the loop.
Deallocating Odd-Indexed Elements and Freeing Memory: The for loop iterates over the odd-indexed elements of AR1, AR2,
and AR3, and another for loop iterates over the elements of AR4, performing memory deallocation (free). The time
complexity of the deallocation operation (free) is O(1). Therefore, the time complexity of this section is O(p + q),
where p is the number of odd-indexed elements in AR1, AR2, and AR3, and q is the number of elements in AR4.

Overall, the time complexity of the code can be expressed as O(n + m + k + p + q).

Space Complexity -

The code uses dynamic memory allocation to allocate and deallocate memory blocks. The space complexity mainly depends on the maximum number of memory blocks allocated simultaneously, which is determined by the available memory and the element sizes.

In this code, there are four arrays: AR1, AR2, AR3, and AR4. The space complexity is determined by the total size of these arrays and the memory blocks they hold. The size of each memory block is element_size or ar4_element_size, and the number of blocks is determined by the respective counts (count and ar4_count).

Therefore, the space complexity of the code can be expressed as O(count * element_size + ar4_count * ar4_element_size), where count is the number of elements in AR1, AR2, and AR3, and ar4_count is the number of elements in AR4.

Note that the actual space usage may also depend on the implementation details of the memory allocation functions (malloc, realloc, free) and the underlying memory management system.

OUTPUT FOR RUN 1:

```
Time taken for initial allocation: 22 seconds
Number of elements in AR1, AR2, AR3: 130001
Time taken for deallocation of even number elements: 181 milliseconds
Time taken for allocation to AR4: 177 seconds
Number of elements in AR4: 13000
```

OUTPUT FOR RUN 2:

```
Time taken for initial allocation: 17 seconds
Number of elements in AR1, AR2, AR3: 130001
Time taken for deallocation of even number elements: 147 milliseconds
Time taken for allocation to AR4: 171 seconds
Number of elements in AR4: 13000
```

OUTPUT FOR RUN 3:

```
Time taken for initial allocation: 17 seconds
Number of elements in AR1, AR2, AR3: 130001
Time taken for deallocation of even number elements: 140 milliseconds
Time taken for allocation to AR4: 170 seconds
Number of elements in AR4: 13000
```

To compile and run the code, you can use the following commands:

```
gcc Assignment5.c -o Assignment
```

```
./Assignment

*/
```

PROGRAM 6

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>

using namespace std;
using namespace chrono;

// Function to fill the array with random values
void fill(vector<int>& arr);

// Binary search function
int binarySearch(const std::vector<int>& arr, int target);

int main() {
    // Initialize arrays of varying sizes
    vector<vector<int>> arrays = {{100}, {200}, {400}, {800}, {1600}, {3200},
                                  {6400}, {12800}, {25600}, {51200}};

    int index = 0;
    int target = 2147483647;

    vector<milliseconds> durations(arrays.size());

    // Perform binary search on each array and measure the execution time
    for (int i = 0; i < arrays.size(); i++) {
        vector<int>& arr = arrays[i];
        auto start = high_resolution_clock::now();
        for (int j = 0; j < 10000000; j++)
            index = binarySearch(arr, target);
        auto end = high_resolution_clock::now();
        durations[i] = duration_cast<milliseconds>(end - start);
    }

    // Output the durations
    for (int i = 0; i < arrays.size(); i++) {
        cout << "Time " << i + 1 << ": " << durations[i].count() << "
milliseconds" << endl;
    }
```

```
    return 0;
}

//initializes the vector of arrays
void fill(vector<int>& arr){
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dis(1, 1000);
    for (int i = 0; i < arr.size(); i++){
        arr[i] = dis(gen);
    }
}

//Binary search function for C++
int binarySearch(const std::vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return -1;
}

/*
DATA STRUCTURES USED - Vectors, vector of vectors

The given code performs binary search on different arrays of varying sizes and
measures the
execution time for each search operation. The code uses the binarySearch function
to perform
the binary search algorithm. It initializes a vector of vectors (arrays) to hold
the different
```

arrays to be searched. The code then iterates over each array, performs 10,000,000 unsuccessful
search operations, and records the duration of each search. The durations are stored in a
separate vector (durations) and finally displayed on the console.

Time Complexity:
The time complexity of the binary search algorithm is O(log n), where n is the size of the
input array. In this code, binary search is performed 10,000,000 times for each array in the
arrays vector. Therefore, the time complexity of the code is O(10,000,000 * log n), where n
represents the size of each array.

Space Complexity:
The space complexity of the code is determined by the space used to store the arrays and the
durations vector. The space required for the arrays is proportional to the total number of
elements across all arrays. The space required for the durations vector is proportional to
the number of arrays. Therefore, the space complexity can be considered as O(N), where N is
the total number of elements across all arrays plus the number of arrays

Run 1::
Time 1: 89 milliseconds
Time 2: 89 milliseconds
Time 3: 89 milliseconds
Time 4: 88 milliseconds
Time 5: 88 milliseconds
Time 6: 90 milliseconds
Time 7: 90 milliseconds
Time 8: 89 milliseconds
Time 9: 89 milliseconds
Time 10: 91 milliseconds

Run 2::
Time 1: 91 milliseconds
Time 2: 91 milliseconds
Time 3: 91 milliseconds
Time 4: 89 milliseconds
Time 5: 97 milliseconds
Time 6: 109 milliseconds

```
Time 7: 97 milliseconds
Time 8: 96 milliseconds
Time 9: 89 milliseconds
Time 10: 97 milliseconds

Run 3::
Time 1: 89 milliseconds
Time 2: 89 milliseconds
Time 3: 95 milliseconds
Time 4: 91 milliseconds
Time 5: 88 milliseconds
Time 6: 88 milliseconds
Time 7: 94 milliseconds
Time 8: 90 milliseconds
Time 9: 89 milliseconds
Time 10: 91 milliseconds
*/


import random
import time

def fill(arr): #initialize arrays
    for i in range(len(arr)):
        arr[i] = random.randint(1, 1000)

def binary_search(arr, target): #binary search function
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

arrays = [[100], [200], [400], [800], [1600], [3200],
          [6400], [12800], [25600], [51200]] #all arrays doubling until 51200
```

```
target = 2147483647 #target that we won't find

durations = []

#initialize array
for arr in arrays: #main
    fill(arr)
    start = time.perf_counter()
    for _ in range(10000000):
        index = binary_search(arr, target)
    end = time.perf_counter()
    durations.append((end - start))  # In seconds

#print out timings
for i, duration in enumerate(durations):
    print(f"Time {i + 1}: {duration} seconds")

"""
DATA STRUCTURES USED - Lists

The given code performs binary search on different arrays of varying sizes and
measures
the execution time for each search operation. It uses the fill function to
initialize
the arrays with random values and the binary_search function to perform the
binary search
algorithm. The arrays are defined as a list of lists (arrays), where each inner
list
represents an array with a specific size. The code iterates over each array,
performs
10,000,000 search operations for each array, and records the duration of each
search.
The durations are stored in a separate list (durations) and then displayed on the
console.

Time Complexity:
The time complexity of the binary search algorithm is O(log n), where n is the
size of the
input array. In this code, binary search is performed 10,000,000 times for each
array in the
arrays list. Therefore, the time complexity of the code is O(10,000,000 * log n),
where n
represents the size of each array.

Space Complexity:
```

The space complexity of the code is determined by the space used to store the arrays and
the durations list. The space required for the arrays is proportional to the total number
of elements across all arrays. The space required for the durations list is proportional
to the number of arrays. Therefore, the space complexity can be considered as O(N), where
N is the total number of elements across all arrays plus the number of arrays.

Run 1::
Time 1: 4.6877584 seconds
Time 2: 4.8131024 seconds
Time 3: 4.7948068 seconds
Time 4: 4.866584999999999 seconds
Time 5: 4.7196494000000015 seconds
Time 6: 4.6712983 seconds
Time 7: 4.765804000000003 seconds
Time 8: 4.740078799999999 seconds
Time 9: 4.805248899999995 seconds
Time 10: 4.699169300000001 seconds

Run2::
Time 1: 4.7471711 seconds
Time 2: 4.8277282999999995 seconds
Time 3: 4.757702399999999 seconds
Time 4: 4.861356799999999 seconds
Time 5: 4.986174000000002 seconds
Time 6: 4.759046100000003 seconds
Time 7: 4.805968200000002 seconds
Time 8: 4.853549100000002 seconds
Time 9: 4.789194599999995 seconds
Time 10: 4.840418200000002 seconds

Run3::
Time 1: 4.8334048 seconds
Time 2: 4.7744188 seconds
Time 3: 4.8920178 seconds
Time 4: 4.7807466000000005 seconds
Time 5: 4.808437699999999 seconds
Time 6: 5.002357700000001 seconds
Time 7: 4.862001900000003 seconds
Time 8: 4.741033300000005 seconds
Time 9: 4.766355500000003 seconds
Time 10: 4.8410054 seconds

```
"""
use rand::Rng; //random
use std::time::Instant; //time measurement

fn fill(arr: &mut [i32]) { //fill array with mutible ints with 32 bits
    let mut rng = rand::thread_rng();
    for i in arr.iter_mut() {
        *i = rng.gen_range(1..=1000);
    }
}

fn binary_search(arr: &[i32], target: i32) -> Option<usize> { //binary search
function
    let mut left = 0;
    let mut right = arr.len() - 1;

    while left <= right {
        let mid = left + (right - left) / 2;

        if arr[mid] == target {
            return Some(mid);
        } else if arr[mid] < target {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    None
}

fn main() {
    let arrays = vec![ //mutable arrays
        vec![0; 100],
        vec![0; 200],
        vec![0; 400],
        vec![0; 800],
        vec![0; 1600],
        vec![0; 3200],
        vec![0; 6400],
        vec![0; 12800],
        vec![0; 25600],
        vec![0; 51200],
    ];
```

```
    let target = 2_147_483_647; //impossible number to find

    let mut durations = Vec::new(); //time measurement

    for arr in &arrays { //main
        fill(&mut arr.clone());
        let start = Instant::now();
        for _ in 0..10_000_000 {
            let _ = binary_search(&arr, target);
        }
        let end = Instant::now();
        let duration = end.duration_since(start).as_nanos();
        durations.push(duration);
    }

    for (i, duration) in durations.iter().enumerate() {
        println!("Time {}: {} nanoseconds", i + 1, duration);
    }
}

/*
DATA STRUCTURES USED - Vectors, Arrays

The provided code performs binary search on different mutable arrays of varying
sizes and
measures the execution time for each search operation. It uses the fill function
to populate
the arrays with random integers and the binary_search function to perform the
binary
search algorithm. The arrays are defined as a vector of mutable vectors (arrays).
The
code iterates over each array, fills it with random values, performs 10,000,000
search
operations on each array, and records the duration of each search. The durations
are stored
in a separate vector (durations) and then displayed on the console.

Time Complexity:
The time complexity of the binary search algorithm is O(log n), where n is the
size of the
input array. In this code, binary search is performed 10,000,000 times for each
array in
the arrays vector. Therefore, the time complexity of the code can be considered
as
O(10,000,000 * log n), where n represents the size of each array.
```

Space Complexity:
The space complexity of the code is determined by the space used to store the arrays and the
durations vector. The space required for the arrays is proportional to the total number of
elements across all arrays. The space required for the durations vector is proportional to
the number of arrays. Therefore, the space complexity can be considered as O(N), where N
is the total number of elements across all arrays plus the number of arrays.

Run 1::
Time 1: 1.1925203 seconds
Time 2: 1.4850099 seconds
Time 3: 1.5953257 seconds
Time 4: 1.7131715 seconds
Time 5: 2.0921714 seconds
Time 6: 2.8639674 seconds
Time 7: 2.437687 seconds
Time 8: 2.4179654 seconds
Time 9: 2.577816 seconds
Time 10: 2.7117438 seconds

Run2::
Time 1: 1.6071248 seconds
Time 2: 1.4526246 seconds
Time 3: 1.7735745 seconds
Time 4: 1.9741895 seconds
Time 5: 1.9435543 seconds
Time 6: 2.2024632 seconds
Time 7: 2.1590178 seconds
Time 8: 2.1225288 seconds
Time 9: 2.5231159 seconds
Time 10: 2.7715585 seconds

Run3::
Time 1: 1.5025101 seconds
Time 2: 1.5018165 seconds
Time 3: 2.18645 seconds
Time 4: 2.513827 seconds
Time 5: 2.2210145 seconds
Time 6: 2.2992594 seconds
Time 7: 2.240435 seconds
Time 8: 2.5571263 seconds

```
Time 9: 2.4283183 seconds
Time 10: 2.4862328 seconds
*/
```

PROGRAM 7

```
#include <iostream>
#include <queue>
#include <unordered_map>
using namespace std;

// Huffman tree node
struct Node {
    char letter;
    unsigned count;
    Node* left, * right;

    Node(char letter, unsigned count) : letter(letter), count(count),
left(nullptr), right(nullptr) {}
};

// Comparison operator for priority queue
struct compare {
    bool operator()(Node* l, Node* r) {
        return l->count > r->count;
    }
};

// Build Huffman tree and generate codes
unordered_map<char, string> buildHuffmanTree(string text) {
    // Count the frequency of each character
    unordered_map<char, unsigned> frequencies;
    for (char c : text) {
        frequencies[c]++;
    }

    // Create a priority queue (min-heap) to store the nodes of the Huffman tree
    priority_queue<Node*, vector<Node*>, compare> pq;
    for (auto pair : frequencies) {
        pq.push(new Node(pair.first, pair.second));
    }

    // Build the Huffman tree
    while (pq.size() > 1) {
        Node* left = pq.top();
```

```
        pq.pop();
        Node* right = pq.top();
        pq.pop();

        Node* newNode = new Node('$', left->count + right->count);
        newNode->left = left;
        newNode->right = right;

        pq.push(newNode);
    }

    // Generate Huffman codes
    unordered_map<char, string> codes;
    if (!pq.empty()) {
        Node* root = pq.top();
        string currentCode = "";
        // Traverse the Huffman tree and assign codes
        queue<pair<Node*, string>> nodeQueue;
        nodeQueue.push(make_pair(root, currentCode));

        while (!nodeQueue.empty()) {
            Node* node = nodeQueue.front().first;
            currentCode = nodeQueue.front().second;
            nodeQueue.pop();

            if (!node->left && !node->right) {
                codes[node->letter] = currentCode;
            }

            if (node->left) {
                nodeQueue.push(make_pair(node->left, currentCode + "0"));
            }

            if (node->right) {
                nodeQueue.push(make_pair(node->right, currentCode + "1"));
            }
        }
    }

    return codes;
}

int main() {
    string text = "abcde";
```

```
    unordered_map<char, string> codes = buildHuffmanTree(text);
    // Display the generated Huffman codes
    cout << "Huffman Codes:" << endl;
    for (auto pair : codes) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}

/*
The provided code implements the Huffman coding algorithm to build a Huffman tree and generate
Huffman codes for characters in a given text. The code defines a Node struct representing a node
in the Huffman tree, and a compare struct as the comparison operator for the priority queue used
to build the tree. The buildHuffmanTree function takes a string as input, counts the frequency
of each character in the text, creates a priority queue of nodes based on the frequencies, builds
the Huffman tree using the priority queue, and generates Huffman codes for each character. The
generated codes are stored in an unordered map. Finally, the generated Huffman codes are displayed
on the console.

Time Complexity:

Counting the frequency of each character in the input text takes O(n) time, where n is the
length of the text.
Building the Huffman tree involves iterating over the frequencies and performing operations on
the priority queue. Each iteration takes O(log n) time, where n is the number of distinct characters
in the text. Therefore, building the Huffman tree has a time complexity of O(n log n) in the worst case.
Generating Huffman codes involves traversing the Huffman tree, which has a total of n nodes (n is
the number of distinct characters). The traversal operation takes O(n) time, as each node is visited
exactly once. Thus, generating the Huffman codes has a time complexity of O(n).
Overall, the time complexity of the buildHuffmanTree function is O(n log n).
```

Space Complexity:

The space required for storing the characters and their frequencies in the unordered map is proportional
to the number of distinct characters in the text, which can be at most O(n), where n is the length of the text.
The priority queue stores the nodes of the Huffman tree. The maximum number of nodes in the tree is equal
to the number of distinct characters, which can be at most O(n).
The generated Huffman codes are stored in an unordered map, which requires space proportional to the number
of distinct characters, i.e., O(n).
In addition to the input text, the space complexity is dominated by the Huffman tree, which is proportional
to the number of distinct characters, i.e., O(n).
Overall, the space complexity of the code is O(n).

*/