# Assignment-1

**Name** – Ujjwal Lehri
**Roll No.** – 60216403224
**Course** – B.Tech CSE 4th sem

---

**Q. Write a program in C/C++, show its output, plot the graph, and calculate it's time complexity for**
**1. Direct Search**
**2. Linear Search**
**3. Binary Search**

**Code:**

```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>

typedef struct Block {
   int key;
   struct Block* next;
} Block;


/*************************************************************************/

int* GenArr(int size) {
   int* Arr = (int*)malloc(sizeof(int) * size);
   for (int i = 0; i < size; i++) {
      Arr[i] = i;
   }
   return Arr;
}
```

```
/*****************************************************************************/
/*                                  Linear Search                            */
/*****************************************************************************/

int LinearSearch(int target, int *Arr, int size) {
   for (unsigned int i = 0; i < size; i++) {
      if (Arr[i] == target) return i;
   }
   return -1;
}


/*****************************************************************************/
/*                                  Binary Search                            */
/*****************************************************************************/

int BinarySearch(int *Arr, int target, int start, int end) {
   if (start > end) {
      return end + 1;
   }

   int middle = (start + end) / 2;
   if (Arr[middle] == target) {
      return middle;
   } else if (Arr[middle] > target) {
      return BinarySearch(Arr, target, start, middle - 1);
   } else {
      return BinarySearch(Arr, target, middle + 1, end);
   }
}


/*****************************************************************************/
/*                                  Direct Search                            */
/*****************************************************************************/

int hashFunction(int key, int SIZE) {
   double fraction = key * 0.6180339887;
   fraction = fraction - (long long)fraction;
   return (int)(SIZE * fraction);
}

/*****************************************************************************/
```

```
Block* createBlock(int key) {
   Block* node = (Block*)malloc(sizeof(Block));
   node->key = key;
   node->next = NULL;
   return node;
}

/*************************************************************************/


void put(Block** hashTable, int key, int SIZE) {
   int index = hashFunction(key, SIZE);
   Block* node = createBlock(key);
   node->next = hashTable[index];
   hashTable[index] = node;
}

/*************************************************************************/

int search(Block** hashTable, int key, int SIZE) {
   int index = hashFunction(key, SIZE);
   Block* temp = hashTable[index];
   while (temp != NULL) {
      if (temp->key == key) {
         return index;
      }
      temp = temp->next;
   }
   return -1;
}
```

```
/*****************************************************************************/
/*                                    Main                                   */
/*****************************************************************************/

int main() {
   LARGE_INTEGER frequency, start, end;
   QueryPerformanceFrequency(&frequency);

   int Sizes[] = {100, 1000, 10000, 100000, 1000000};
   int n = sizeof(Sizes) / sizeof(Sizes[0]);

   //---------------------------------------------------------------------------
   printf("Linear Search:\n");
   printf("Input size \t   time taken\n");

   for (int i = 0; i < n; i++) {
      int s = Sizes[i];
      int* Arr = GenArr(s);
      int target = s - 1;
      QueryPerformanceCounter(&start);
      int index = LinearSearch(target, Arr, s);
      QueryPerformanceCounter(&end);
      double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / frequency.QuadPart;
      printf("%d  %20.2lf ns\n", s, time_taken);
      free(Arr);
   }

   //---------------------------------------------------------------------------
   printf("\n\nBinary Search:\n");
   printf("Input size \t     time taken\n");
   for (int i = 0; i < n; i++) {
      int s = Sizes[i];
      int* Arr = GenArr(s);
      int target = s - 1;
      QueryPerformanceCounter(&start);
      int index = BinarySearch(Arr, target, 0, s);
      QueryPerformanceCounter(&end);

      double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / frequency.QuadPart;
      printf("%-7d  %20.2lf ns\n", s, time_taken);
      free(Arr);
   }
```

```
    //-------------------------------------------------------------------------------
    printf("\n\nDirect Search (Hash Table):\n");
    printf("Input size \t     time taken\n");

    for (int i = 0; i < n; i++) {
        int s = Sizes[i];

        Block** hashTable = (Block**)malloc(sizeof(Block*) * s);
        for (int j = 0; j < s; j++) {
            hashTable[j] = NULL;
        }

        int* Arr = GenArr(s);
        int target = s - 1;

        for (int j = 0; j < s; j++) {
            put(hashTable, Arr[j], s);
        }

        QueryPerformanceCounter(&start);
        int index = search(hashTable, target, s);
        QueryPerformanceCounter(&end);

        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / frequency.QuadPart;
        printf("%-7d  %20.2lf ns\n", s, time_taken);

        free(Arr);
        free(hashTable);
    }

    return 0;
}
```

**Output**:

```
PS C:\Users\Ujjwal\Desktop\C\Search_algo> cd "c:\Users\Ujjwal\Desktop\C\Search_algo\" ; if ($?) { gcc Search.c -o Search } ; if ($?) { .\Search }
Linear Search:
Input size        time taken
100               800.00 ns
1000              3800.00 ns
10000             35900.00 ns
100000            319600.00 ns
1000000           3240200.00 ns


Binary Search:
Input size          time taken
100                  600.00 ns
1000                 600.00 ns
10000                500.00 ns
100000               700.00 ns
1000000             1400.00 ns


Direct Search (Hash Table):
Input size          time taken
100                  500.00 ns
1000                 300.00 ns
10000                300.00 ns
100000               300.00 ns
1000000              300.00 ns
```

**Graph**: the graph compares the performance of Direct Search (using hashing), Linear Search, and Binary Search across different array sizes.