# ALGORITHM LABORATORY

## ASSIGNMENT-1

**PROBLEM STATEMENT:** Given a set of records $R_1$, ...., $R_n$ identified by keys $K_1$, ...., $K_n$ and a key K, decide whether the record corresponding to key K exists or not.

**SCENARIOS:** Write a program to implement the searching algorithms for each of the following situations:

1. *The input records are not sorted based on the keys and each key has equal probability of getting searched.*

### ALGORITHM (LINEAR SEARCH):

a. Start from the first element of the array.
b. Compare the current element with the key to be searched.
c. If a match is found, return the index of the element.
d. If the end of the array is reached without finding the key, return -1 (indicating key not found).
e. The algorithm terminates after either finding the key or scanning all elements.

### PROGRAM CODE:

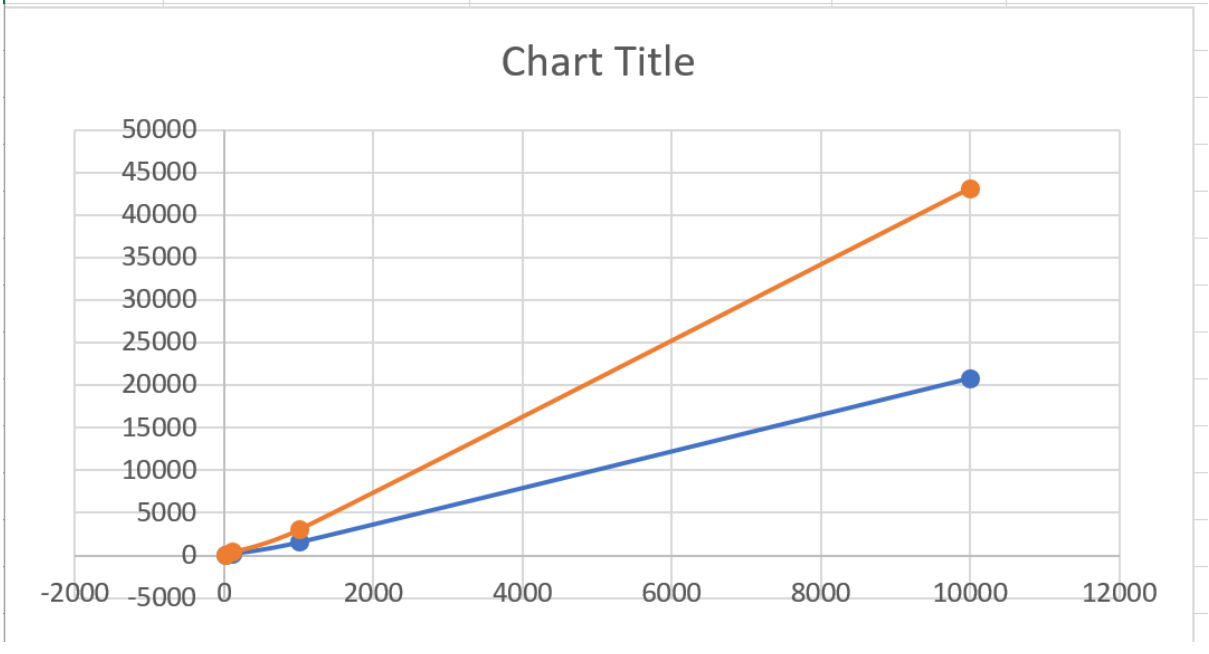
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

pair<int,int> linearSearch(vector<int> &records, int key,int n) {
    int comp = 0;
    comp = 2;
    for (int i = 0; i < n; i++) {
        if (key == records[i]) {
            return {i,comp};
        }
```

```
        comp+=2;
    }
    return {-1,comp};
}

int main() {
    srand(time(NULL));
    for (int k = 10; k <= 10000; k *= 10) {
        unordered_set<int> records;
        unordered_set<int> search_string;

        while (records.size() < k) {
            int random = rand() % (5 * k);
            records.insert(random);
            search_string.insert(random);
        }

        while (search_string.size() < ((3 * k) >> 1)) {
            int random = (5 * k) + (rand() % (5 * k));
            search_string.insert(random);
        }

        vector<int> record(records.begin(), records.end());
        vector<int> search_strings(search_string.begin(),
search_string.end());
        random_shuffle(search_strings.begin(), search_strings.end());

        double time_key_found = 0;
        double avg_comparisions = 0;
        double time_key_not_found = 0;
        int key_found = 0, key_not_found = 0;
        int num_trials = 0;
        if(k<=100){
            num_trials=100000;
        }
        else if(k==1000) num_trials = 100;
        else num_trials = 10;

        for (int trial = 0; trial < num_trials; trial++) {
            for(int i=0;i<search_strings.size();i+=1){
                record.push_back(search_strings[i]);
                auto start = high_resolution_clock::now();
                pair<int,int> p;
                p= linearSearch(record, search_strings[i],k);
                int f = p.first;
                auto end = high_resolution_clock::now();
                record.pop_back();
                auto time = duration_cast<nanoseconds>(end - start);
```

```
                avg_comparisions+=p.second;
                if (f == -1) {
                    time_key_not_found += time.count();
                    key_not_found++;
                } else {
                    time_key_found += time.count();
                    key_found++;
                }
            }
        }
        avg_comparisions/=(key_not_found+key_found);
        double avg_key_found = (key_found > 0) ? time_key_found / key_found :
0;
        double avg_key_not_found = (key_not_found > 0) ? time_key_not_found /
key_not_found : 0;

        cout << "N = " << k << " | Key Found Avg = " << avg_key_found << "
nanoseconds"
        << " | Key Not Found Avg = " << avg_key_not_found << " nanoseconds" <<
" | Avg_num_of_comparisions = "<<avg_comparisions<<endl;
    }
    return 0;
}
```

## OUTPUT and PLOT:

| N value | Time key found (ns) | Time key not found (ns) | Avg_comp |
|---|---|---|---|
| 10 | 110.788 | 99.2362 | 14.6667 |
| 100 | 230.512 | 400.489 | 134.667 |
| 1000 | 1577.98 | 3053.93 | 1334.67 |
| 10000 | 20777.5 | 43102.7 | 13334.7 |



2. *The input records are not sorted based on the keys, but each key,* $K_i$, $1 \le i \le n-1$, *has a probability* $p_i$ *of getting searched.*

**ALGORITHM (LINEAR SEARCH):**

a. Start from the first element of the array.
b. Compare the current element with the key to be searched.
c. If a match is found, return the index of the element.
d. If the end of the array is reached without finding the key, return -1 (indicating key not found).
e. The algorithm terminates after either finding the key or scanning all elements.

**PROGRAM CODE:**

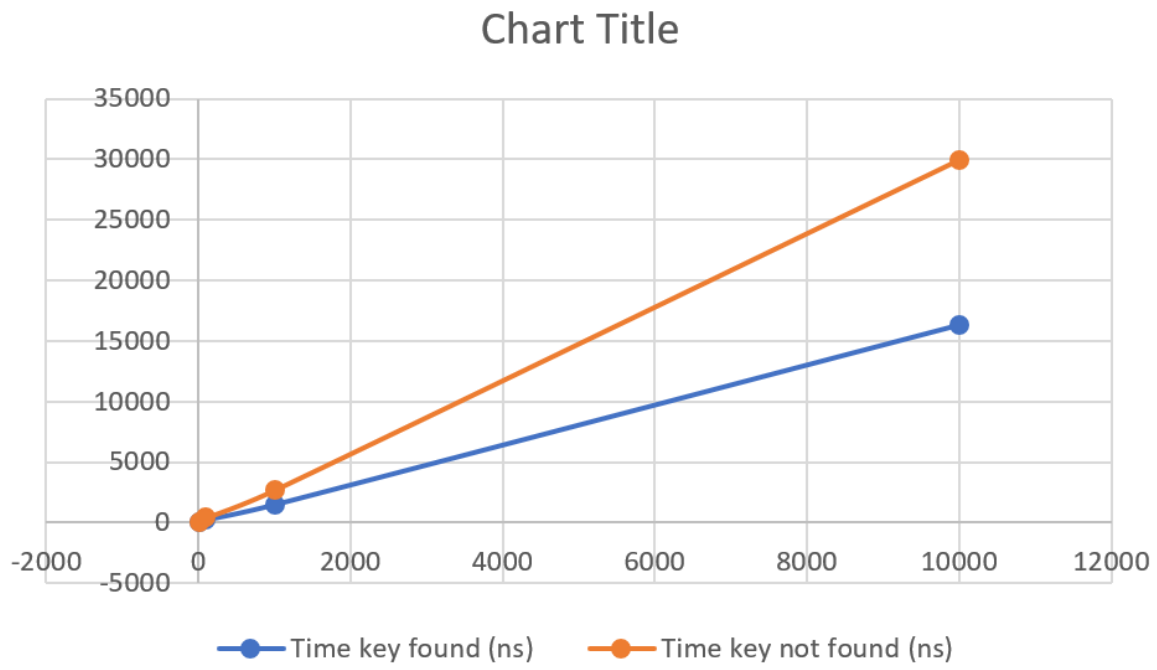
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

pair<int,int> linearSearch(vector<int> &records, int key,int n) {
    int comp = 0;
    comp = 2;
    for (int i = 0; i < n; i++) {
        if (key == records[i]) {
            return {i,comp};
        }
        comp+=2;
    }
    return {-1,comp};
}

int main() {
    srand(time(NULL));
    for (int k = 10; k <= 10000; k *= 10) {
        unordered_set<int> records;
        unordered_set<int> search_string;

        while (records.size() < k) {
            int random = rand() % (5 * k);
            records.insert(random);
            search_string.insert(random);
        }

        while (search_string.size() < ((3 * k) >> 1)) {
```

```
            int random = (5 * k) + (rand() % (5 * k));
            search_string.insert(random);
        }

        vector<int> record(records.begin(), records.end());
        vector<int> search_strings(search_string.begin(),
search_string.end());
        random_shuffle(search_strings.begin(), search_strings.end());

        double time_key_found = 0;
        double avg_comparisions = 0;
        double time_key_not_found = 0;
        int key_found = 0, key_not_found = 0;
        int num_trials = 0;
        if(k<=100){
            num_trials=100000;
        }
        else if(k==1000) num_trials = 100;
        else num_trials = 10;

        for (int trial = 0; trial < num_trials; trial++) {
            for(int i=0;i<search_strings.size();i+=1){
                record.push_back(search_strings[i]);
                auto start = high_resolution_clock::now();
                pair<int,int> p;
                p= linearSearch(record, search_strings[i],k);
                int f = p.first;
                auto end = high_resolution_clock::now();
                record.pop_back();
                auto time = duration_cast<nanoseconds>(end - start);
                avg_comparisions+=p.second;
                if (f == -1) {
                    time_key_not_found += time.count();
                    key_not_found++;
                } else {
                    time_key_found += time.count();
                    key_found++;
                }
            }
        }
        avg_comparisions/=(key_not_found+key_found);
        double avg_key_found = (key_found > 0) ? time_key_found / key_found :
0;
        double avg_key_not_found = (key_not_found > 0) ? time_key_not_found /
key_not_found : 0;

        cout << "N = " << k << " | Key Found Avg = " << avg_key_found << "
nanoseconds"
```

```
        << " | Key Not Found Avg = " << avg_key_not_found << " nanoseconds" <<
" | Avg_num_of_comparisions = "<<avg_comparisions<<endl;
    }
    return 0;
}
```

## OUTPUT and PLOT:

| N value | Time key found (ns) | Time key not found (ns) | Avg_comp |
|---|---|---|---|
| 10 | 62.9787 | 73.2813 | 14.4 |
| 100 | 248.572 | 411.016 | 133.787 |
| 1000 | 1499.23 | 2685.66 | 1326.35 |
| 10000 | 16367.5 | 29984.5 | 13261.4 |



3. *The input records are sorted based on keys.*

## ALGORITHM (BINARY SEARCH):

f. **Initialize pointers**:
   i. Set low = 0 (starting index).
   ii. Set high = n - 1 (ending index), where n is the number of records.

g. **Repeat the following steps while low <= high:**
   i. Calculate the middle index: mid = (low + high) / 2.
   ii. Compare the middle element with the key:
      1. If the middle element is **equal** to the key, return the index mid (key found).
      2. If the middle element is **less than** the key, set low=mid+1 (search in the right half).
      3. If the middle element is **greater than** the key, set high=mid - 1 (search in the left half).

h. **End the search**:
   i. If the key is not found and low > high, return -1 (key not found).

## PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

pair<int,int> binarySearch(vector<int> &records, int key) {
    int comp = 0;
    int low = 0;
    int high = records.size() - 1;
    while (low <= high) {
        comp++;
        int mid = (low + high) >> 1;
        if (records[mid] == key) return {mid,comp};
        else if (records[mid] > key) high = mid - 1;
        else low = mid + 1;
    }
    return {-1,comp};
}

int main() {
```

```
    srand(time(NULL));

    for (int k = 10; k <= 10000; k *= 10) {
        set<int> records;
        unordered_set<int> search_string;

        while (records.size() < k) {
            int random = rand() % (5 * k);
            records.insert(random);
            search_string.insert(random);
        }

        while (search_string.size() < ((3 * k) >> 1)) {
            int random = (5 * k) + (rand() % (5 * k));
            search_string.insert(random);
        }

        vector<int> record(records.begin(), records.end());
        vector<int> search_strings(search_string.begin(),
search_string.end());
        random_shuffle(search_strings.begin(), search_strings.end());

        double time_key_found = 0;
        double time_key_not_found = 0;
        int key_found = 0, key_not_found = 0;
        double avg_comp = 0;

        int num_trials = (k == 10000) ? 1000 : (k == 1000) ? 10000 : 100000;

        for (int trial = 0; trial < num_trials; trial++) {
            for (int i = 0; i < search_strings.size(); i++) {
                auto start = high_resolution_clock::now();
                pair<int,int> p;
                p = binarySearch(record, search_strings[i]);
                auto end = high_resolution_clock::now();
                auto time = duration_cast<nanoseconds>(end - start);
                int f = p.first;
                avg_comp+= p.second;
                if (f == -1) {
                    time_key_not_found += time.count();
                    key_not_found++;
                } else {
                    time_key_found += time.count();
                    key_found++;
                }
            }
        }
```

```
        double avg_key_found = (key_found > 0) ? time_key_found / key_found :
0;
        double avg_key_not_found = (key_not_found > 0) ? time_key_not_found /
key_not_found : 0;
        avg_comp/=(key_found+key_not_found);
        cout << "N = " << k
             << " | Key Found Avg = " << avg_key_found << " nanoseconds"
             << " | Key Not Found Avg = " << avg_key_not_found << "
nanoseconds"
             << " | Avg_comp = "<<avg_comp<<" nanoseconds"
             << endl;
    }
    return 0;
}
```

## OUTPUT and PLOT:

| N value | Time key found (ns) | Time key not found (ns) | Avg_comp |
|---|---|---|---|
| 10 | 67.9453 | 73.5278 | 3.26667 |
| 100 | 97.7511 | 99.2095 | 6.2 |
| 1000 | 144.813 | 124.285 | 9.32467 |
| 10000 | 204.676 | 130.242 | 12.9087 |