ALGORITHM LABORATORY

ASSIGNMENT-1a

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. Searching is done in the following manner.*

ALGORITHM (LINEAR SEARCH):

a. Input: A vector records of size n, a key to search.
b. Push: Insert key at the end of the records vector.
c. While i <= n (search until the end):
    i. Compare records[i] with key.
    ii. If records[i] == key, return i.
    iii. If not, increment i by 1
d. If Not Found: If i == n, return -1 (key not found).
e. Output: Return the index of the key

PROGRAM CODE:

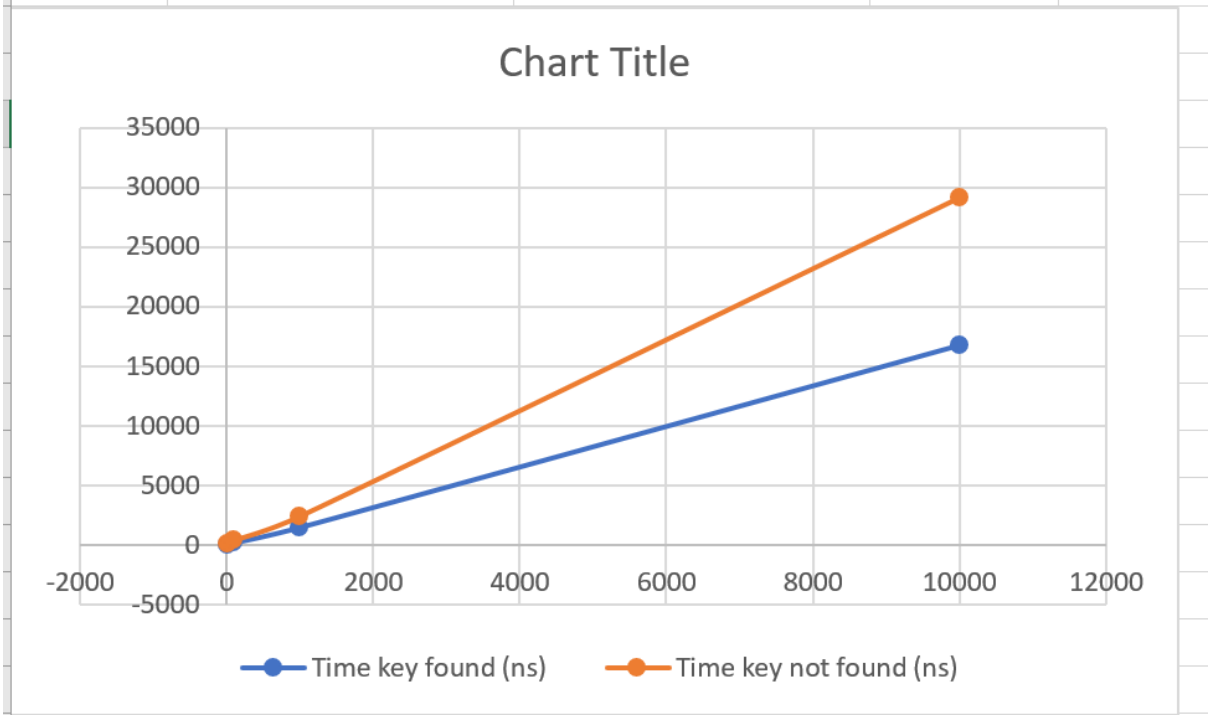
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

pair<int,int> linearSearch(vector<int> &records, int key,int n) {
    int i=0;
    int comp = 1;
    while(records[i]!=key){
        i++;
        comp++;
    }
```

```
    if(i==n) return {-1,comp};
    else return {i,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	77.3	113.152	7.333333
100	249.388	408.644	67.333333
1000	1535.52	2414.96	667.33333
10000	16771.9	29181.7	6667.3333



2. *Each key, K_i, $1 \leq i \leq n-1$,* has a probability $p_i = 1/2^i$ *of getting searched. The input records are sorted in descending order based on the search probability. That is, the maximum searched element is put first, and so on.*

    a. Store probability for each index positions of the primary key array.

    b. Create large streams of search keys maintaining this probability.

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:

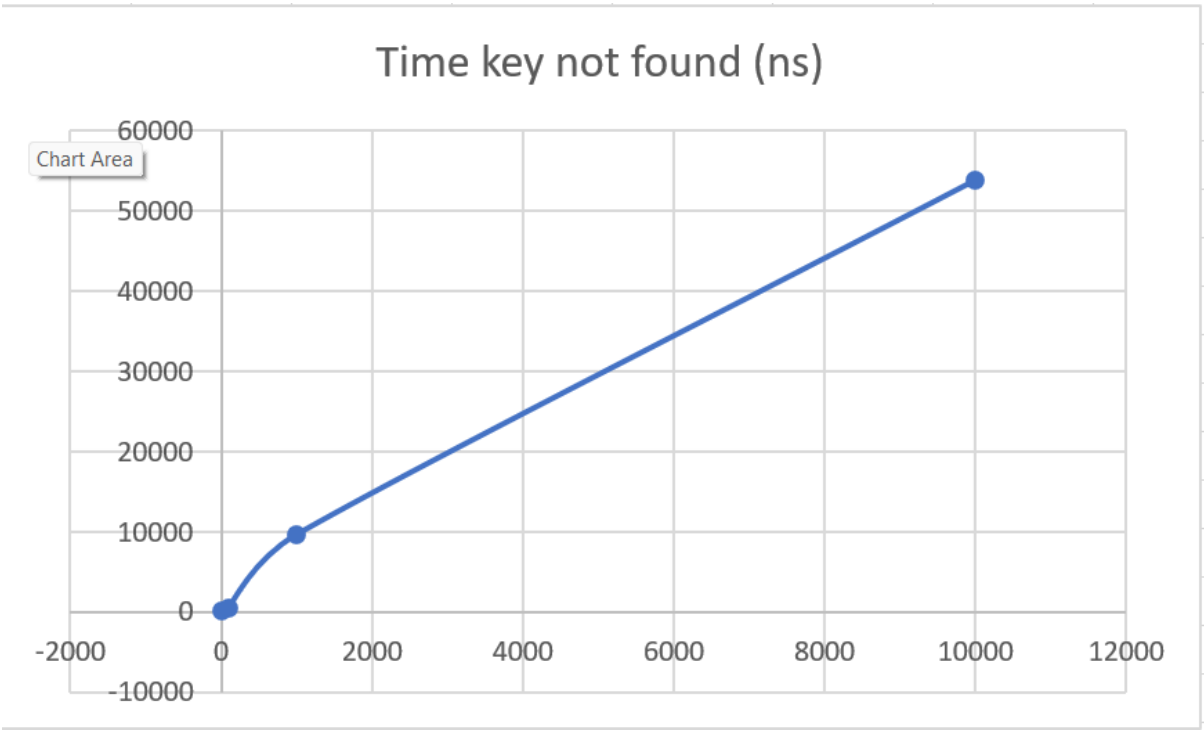
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

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));
    vector<int> sizes = {10,100,1000,10000};
    for(int n : sizes){
        vector<double> probabilities(n);
```

```
        double sum_probabilities = 0;
        for (int i = 0; i < n-1; i++) {
            probabilities[i] = 1.0 / pow(2, i + 1);
            sum_probabilities+=probabilities[i];
        }
        probabilities[n-1] = 1 - sum_probabilities;

        unordered_set<int> record_set;

        while (record_set.size() < n) {
            int random = rand() % (10 * n);
            record_set.insert(random);
        }

        vector<int> records(record_set.begin(), record_set.end());

        vector<int> search_keys;
        for (int i = 0; i < n; i++) {
            int num_occurrences = static_cast<int>(probabilities[i] * 5 * n);
            for (int j = 0; j < num_occurrences; j++) {
                search_keys.push_back(records[i]);
            }
        }

        for(int i=0;i<5*n;i+=1){
            int random = 10*n + (rand()%(10*n));
            search_keys.push_back(random);
        }

        random_shuffle(search_keys.begin(), search_keys.end());

        double time_key_found = 0, time_key_not_found = 0;
        double avg_comparisons = 0;
        int key_found = 0, key_not_found = 0;
        int num_trials = 10;

        for (int trial = 0; trial < num_trials; trial++) {
            for (int key : search_keys) {
                auto start = high_resolution_clock::now();
                pair<int, int> result = linearSearch(records, key, n);
                auto end = high_resolution_clock::now();
                auto duration = duration_cast<nanoseconds>(end - start);
                avg_comparisons += result.second;
                if (result.first == -1) {
                    time_key_not_found += duration.count();
                    key_not_found++;
                } else {
                    time_key_found += duration.count();
```

```
                    key_found++;
                }
            }
        }

        avg_comparisons /= (key_found + key_not_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 = " << n
             << " | Key Found Avg = " << avg_key_found << " nanoseconds"
             << " | Key Not Found Avg = " << avg_key_not_found << "
nanoseconds"
             << " | Avg_comp = "<<avg_comparisons<<" nanoseconds"
             << endl;
    }
    return 0;
}
```

OUTPUT and PLOT:

N value	Time key found (ns)	Time key not found (ns)	Avg_comp
10	57.0915	91.412	13.0722
100	64.1941	511.526	103.533
1000	66.9574	9614.47	1003.49
10000	154.429	53838	10003.6


Time key not found (ns)
Chart Area
60000
50000
40000
30000
20000
10000
0
-10000
-2000
0
2000
4000
6000
8000
10000
12000

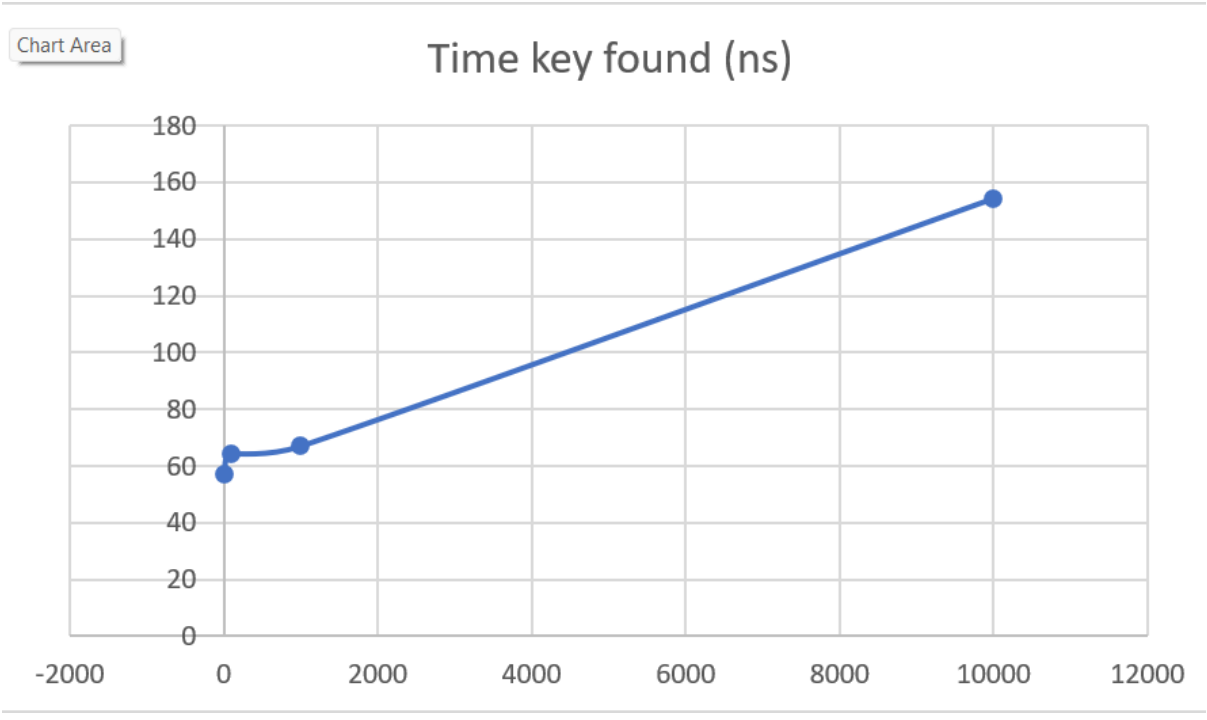


Chart Area
Time key found (ns)
180
160
140
120
100
80
60
40
20
0
-2000
0
2000
4000
6000
8000
10000
12000