106119029, Dipesh Kafle Lab3 Report

You are given a set of records with keyswhich are sorted, but you don't know the total number of records present in the list. Given a search key, design an a: lgorithm such that the time complexity of it is bounded by logn, where n is the number of records present in the set.

Draw the graphs for some random key values as well as for the worst case and check if the algorithm is bounded by logn for each of the cases.

Code

- I have run the search, worst case and random case for sorted records of finite size but the search is run in such a way that its assuming the array size is infinite. So Initially , I start off by doubling each time the right index, until I either encounter an element that is greater than key or I am met with out of bounds error. To check this, I have put the record access inside a try catch block , so that there wont be program termination. Now after doing this, I would have pinpointed the interval in which the key could possibly be present and then I do binay search inside that interval again with appropriate except handling.
- I have made random case to be searching for an element that lies at an index randomly generated by rand() , best case being the first element , and worst case to be the value size. Although the size is mentioned here, inside the search procedure there is no usage whatsoever of the size of the record.
- I have printed values to stdout as well as to a file, which i will open with python and plot the values and get the bounding constant from.

```
#include <algorithm>
#include <chrono>
#include <exception>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std;

int BSearch(vector<int> &vec, int key) {
  int l = 0;
  int r = 1;
  // we'll find the range here, within l to r the key will have to be searched
  // again
  while (true) {
    try {
      if (vec.at(r) <= key) {
        l = r;
        r = 2 * r;
      } else {
        break;
      }
    } catch (std::exception &e) {
      break;
    }
  }
```

```cpp
  int m;
  // normal bseearch on the range we found
  while (l < r) {
    try {
      m = (l + r) / 2;
      if (vec.at(m) == key) {
        return m;
      } else if (vec[m] < key) {
        l = m + 1;
      } else {
        r = m;
      }
    } catch (std::exception &e) {
      r = m;
    }
  }
  return -1;
}
```

```cpp
int main() {
  srand(0);
  vector<int> vec;
  ofstream random_case("RandomCase.txt");
  ofstream worst_case("WorstCase.txt");
  ofstream best_case("BestCase.txt");
  chrono::duration<double> elapsed_time;
  vector<int> sizes(10);
  sizes[0] = 16000;
  for (int i = 1; i < 10; i++)
    sizes[i] = 2 * sizes[i - 1];
  for (int size : sizes) {
    vec.resize(size, 0);
    iota(vec.begin(), vec.end(), 1);
    // Worst Case would be something at the absoulte end of list
    // since our array is 1...size, the size will take worst time
    int key = size;
    auto start_time = chrono::steady_clock::now();
    BSearch(vec, key);
    auto end_time = chrono::steady_clock::now();

    // gets the elapsed time
    elapsed_time =
        chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
    cout << "WORST FOR SIZE " << size << " : " << fixed << setprecision(30)
         << elapsed_time.count() << '\n';
    worst_case << size << ':' << fixed << setprecision(30)
               << elapsed_time.count() << '\n';
```

```cpp
    // random case
    // uses rand()
    key = vec[rand() % size];
    start_time = chrono::steady_clock::now();
    BSearch(vec, key);
    end_time = chrono::steady_clock::now();

    // gets the elapsed time
    elapsed_time =
        chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
    cout << "RANDOM FOR SIZE " << size << " : " << fixed << setprecision(30)
         << elapsed_time.count() << '\n';
    random_case << size << ':' << fixed << setprecision(30)
                << elapsed_time.count() << '\n';

    // best case is index 0
    key = vec[0];
    start_time = chrono::steady_clock::now();
    BSearch(vec, key);
    end_time = chrono::steady_clock::now();

    // gets the elapsed time
    elapsed_time =
        chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
    cout << "BEST FOR SIZE " << size << " : " << fixed << setprecision(30)
         << elapsed_time.count() << '\n';
    best_case << size << ':' << fixed << setprecision(30)
              << elapsed_time.count() << '\n';
  }
}
```

Output

- The first image shows the output printed to stdout by CPP code.
- The second image shows the output printed by python code.

```
WORST FOR SIZE 16000 : 0.000039125999999999997748783426
RANDOM FOR SIZE 16000 : 0.000002429999999999999992180165
BEST FOR SIZE 16000 : 0.000000989999999999999999755228
WORST FOR SIZE 32000 : 0.000020326000000000001564134239
RANDOM FOR SIZE 32000 : 0.000006861999999999999691599126
BEST FOR SIZE 32000 : 0.000000779999999999999996999748
WORST FOR SIZE 64000 : 0.000010989999999999999809731192
RANDOM FOR SIZE 64000 : 0.000007954000000000000417219211
BEST FOR SIZE 64000 : 0.000000649999999999999999705605
WORST FOR SIZE 128000 : 0.000022076000000000000055575336
RANDOM FOR SIZE 128000 : 0.000006214999999999999787580942
BEST FOR SIZE 128000 : 0.000000820000000000000006347861
WORST FOR SIZE 256000 : 0.000020066999999999999469918813
RANDOM FOR SIZE 256000 : 0.000000401000000000000004088608
BEST FOR SIZE 256000 : 0.000000719999999999999996212468
WORST FOR SIZE 512000 : 0.000029518999999999999774881465
RANDOM FOR SIZE 512000 : 0.000004826000000000000042077019
BEST FOR SIZE 512000 : 0.000000108000000000000000936148
WORST FOR SIZE 1024000 : 0.000040775000000000001781214065
RANDOM FOR SIZE 1024000 : 0.000000646000000000000040647498
BEST FOR SIZE 1024000 : 0.000000149999999999999993212217
WORST FOR SIZE 2048000 : 0.000032966000000000002516809677
RANDOM FOR SIZE 2048000 : 0.000000553000000000000416795504
BEST FOR SIZE 2048000 : 0.000000105999999999999996262091
WORST FOR SIZE 4096000 : 0.000034140000000000001779704856
RANDOM FOR SIZE 4096000 : 0.000000513599999999999615120372
BEST FOR SIZE 4096000 : 0.000000105999999999999996262091
WORST FOR SIZE 8192000 : 0.000041402000000000001029686753
RANDOM FOR SIZE 8192000 : 0.000001174999999999999973298811
BEST FOR SIZE 8192000 : 0.000000098999999999999999755228
```

```
λ ~/Acads/Sem4/CSLR41-AlgosLab/Lab3 ➜  python 106119029_plot.py
The upper bound constant for binary search is 2.0422438416969055e-06.
    That means, our binary search was taking at max 2.0422438416969055e-06*log(sizeOfRecord) time
The lower bound constant for binary search is 6.883470178510308e-07.
    That means, our binary search was taking at min6.883470178510308e-07*log(sizeOfRecord) time
```

Plot

- Let our times be [x1,x2,x3..]. Corresponding to those times, let the sizes be [y1,y2,y3..] then what I did is
  - logConstants = [x1/log(y1),x2/log(y2),x3/log(y3)..]
  - logUpperBoundConstant = max(logConstants)
  - logLowerBoundConstant = min(logConstants)
- The upper bound constant for binary search is 2.0422438416969055e-06. That means, our binary search was taking at max 2.0422438416969055e-06*log(sizeOfRecord) time The lower bound constant for binary search is 6.883470178510308e-07. That means, our binary search was taking at min6.883470178510308e-0* 7log(sizeOfRecord) time