# Searching in a sorted array in C

Ying Pei Lin

Fall 2024

## Unsorted Search

The following code search a value in an unsorted array. To prevent the influence of overhead, I made a little change to how the test data is generated. Originally, the test data is generated separately in each iteration. Now, the test data is generated once and the clock get time function is called before and after the for loop that iterates through different array sizes.

```
int main() {
  srand(time(NULL));
  int loop = 1000;
  int array_size[] = {10, 100, 1000, 10000, 100000, 1000000};
  int n = sizeof(array_size) / sizeof(array_size[0]);
  TestData test_data = get_test_data(loop, array_size, n);
  for(int i = 0; i < n; i++) {
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for(int j = 0; j < loop; j++) {
      search(test_data.array_list[i][j], array_size[i], test_data.key_list[i][j]);
    }
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    printf("%d %0.2ld\n", array_size[i], nano_seconds(&t_start, &t_stop)/loop);
  }
}
```

The result is shown in the Figure 1. It shows that the time taken to search linearly in an unsorted array is linearly proportional to the size of the array.

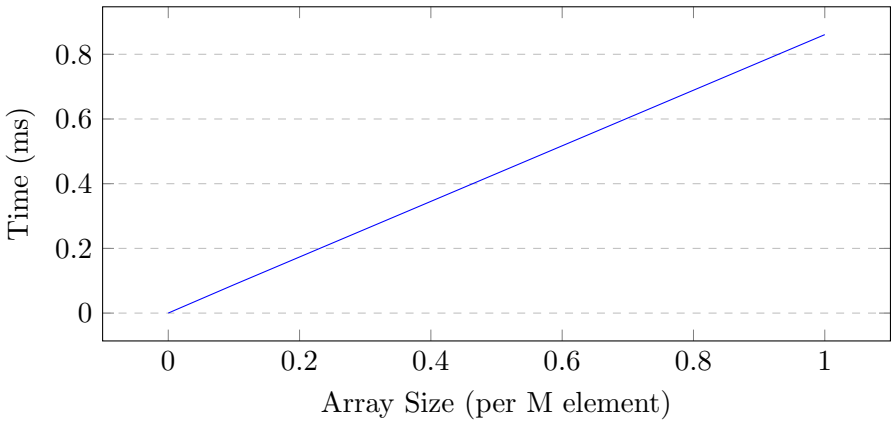


Figure 1: Time taken to search linearly in an unsorted array

## Binary Search

The following code implements the binary seach algorithm. At the beginning, the `first` and `last` index is set to upper and lower bound of the array. The while loop will continue until the key is found or the first index is greater than the last index, which means the key is not found. The `index` is calculated by taking the average of the first and last index. If the value at the index is equal to the key, the function will return true.

If the value at the index is less than the key and the index is less than the last index, which means the key is at the right side of the index. The first index will be updated to index+1, the range of search shrinks to only the right side of the `index`.

If the value at the index is greater than the key and the index is greater than the first index, which means the key is at the left side of the index. The last index will be updated to index-1, the range of search shrinks to only the left side of the `index`.

```
bool binary_search(int array[], int length, int key) {
  int first = 0;
  int last = length-1;
  while (true) {
    int index = (first + last) / 2; // jump to the middle
    if (array[index] == key) {
      return true;
    } else if (array[index] < key && index < last) {
      first = index + 1;
    }else if (array[index] > key && index > first) {
      last = index - 1;
    } else {
```

```
      return false; // Not found
    }
  }
}
```

The result is shown in the Figure 2. The time taken to perform binary search is proportional to the logarithm of the size of the array. This is because the binary search algorithm divides the array into half in each iteration, the number of iteration then will be proportional to $\log_2$(size of the array), namely $O(\log n)$.

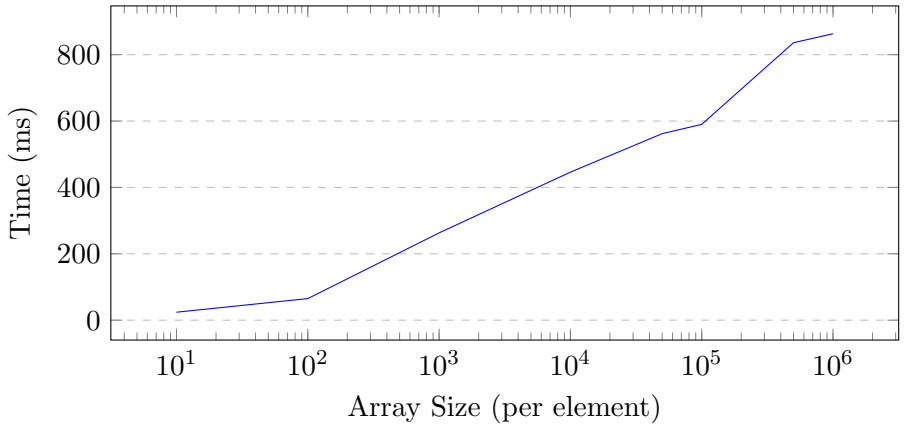


Figure 2: Time taken to search using binary search

To estimate the time it takes to search in an array of size $64M$, I first have to find the regression line that fits the data. I use the following code to fit the data and find the regression function.

```
import numpy as np
import matplotlib.pyplot as plt
data = np.loadtxt('./data/binary_search.dat')
array_size = data[:, 0]
log_array_size = np.log2(array_size)
time_ms = data[:, 1]
a, b = np.polyfit(log_array_size, time_ms, 1)
print(f"Linear regression equation: Time = {a} * log2(ArraySize) + {b}")
```

The result is shown in the Figure 3. The regression line is $Time = 53.17 * \log_2(\text{ArraySize}) - 239.86$. The time taken to search in an array of size $64M$ is $53.17 * \log_2(64M) - 239.86 = 1134.84$ ms $\approx 1100$ ms.

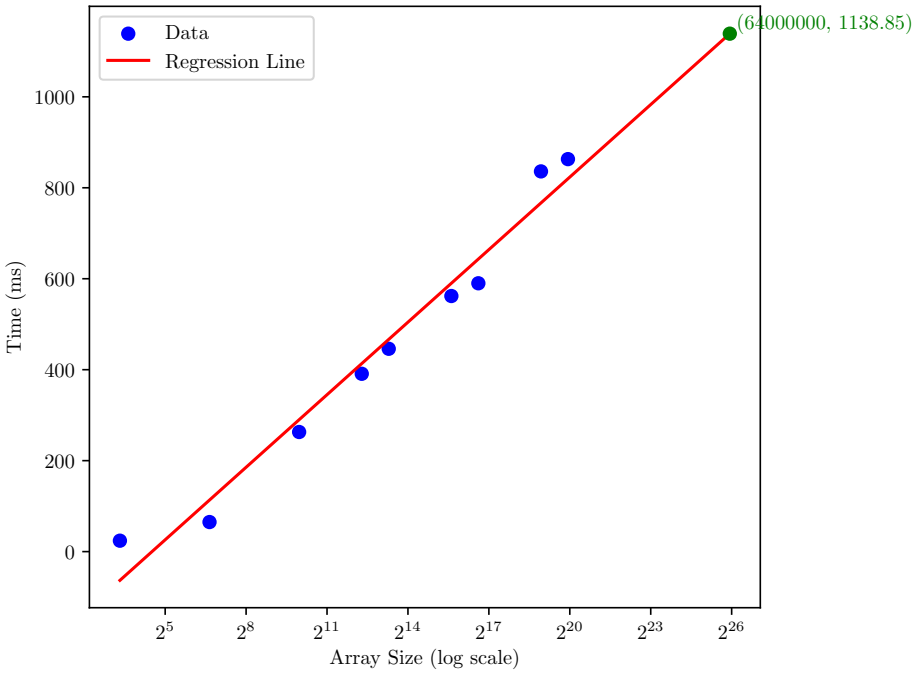


Figure 3: Regression line for binary search

## Recursive Binary Search

The following code implements the recursive binary search algorithm. The difference between the original binary search and the recursive binary search is that the recursive binary search calls itself to search the left or right side of the array while the original binary search uses a while loop to search the array.

```
bool recursive(int array[], int length, int key, int first, int last) {
  int index = (first + last) / 2; // jump to the middle
  if (array[index] == key) {
    return true;
  } else if (array[index] < key && index < last) {
    first = index + 1;
    recursive(array, length, key, first, last);
  } else if (array[index] > key && index > first) {
    last = index - 1;
    recursive(array, length, key, first, last);
  } else {
    return false;
  }
```

```
}
```

The time taken to perform recursive binary search is shown in the Figure 4, which is similar to the binary search.

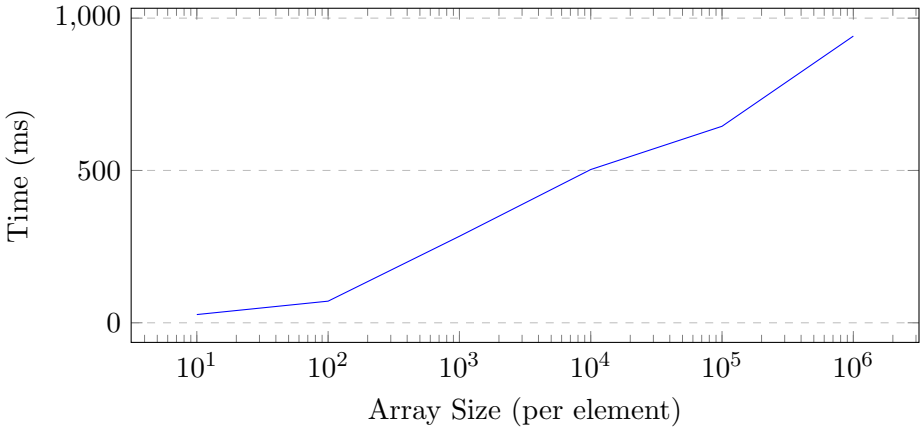


Figure 4: Time taken to search using regression binary search

Though the recursive binary search is more readable and easier to implement, it uses more memory then the original one, because when the function calls itself, it leaves the current function in the stack and enters a new function.

When it is n levels deep, we have n stack frames in the memory. The memory complexity of the recursive binary search is proportional to the depth of the recursion, which is $O(\log n)$ while the original binary search is $O(1)$.

The call times of the recursive binary search is shown in the Figure 5, which is proportional to the logarithm of the size of the array.

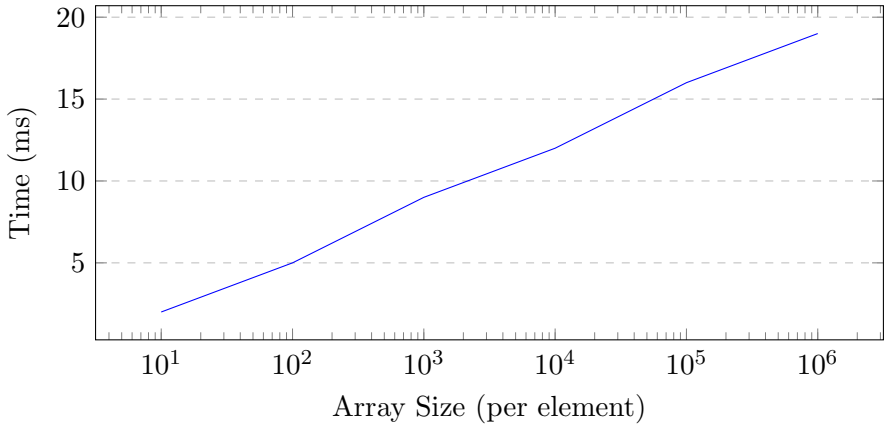


Figure 5: Call times of the recursive binary search