This report presents the Big-O comparison of different search algorithms. This report compares and analyzes the results of a Big-O study focusing on Sequential Search (various kinds) and Binary Search.

BIG-O NOTATION

HOMEWORK # 1

Ayushya Amitabh
CSC 22000 – ALGORITHMS
PROF. CHI HIM “TIMMY” LIU

Contents

THEORY 2

ALGORITHMS IN THIS REPORT 2

HYPOTHESIS 2

STUDYING INDIVIDUAL ARRAY SIZES: 3

- ARRAY OF SIZE 10^2 3
- ARRAY OF SIZE 10^4 3
- ARRAY OF SIZE 10^6 3
- ARRAY OF SIZE 10^8 3

STUDYING OVERALL ALGORITHM PERFORMANCES 4

- ANALYSIS VS HYPOTHESIS 4

LOOKING AT THE CODE 4

- PRE-CODE RESEARCH 4
- GENERATING ARRAYS 5
- STRAIGHT SEQUENTIAL SEARCH 5
- QUICK SORT 6
- ORDERED SEQUENTIAL SEARCH 6
- INDEXED SEQUENTIAL SEARCH 7
- SEQUENTIAL SEARCH WITH FIRST OR PREVIOUS ELEMENT SWAP 8
- BINARY SEARCH 9

APPENDIX: CODE [MAIN.CPP] 10

THEORY

Big O notation is the language we use for articulating how long an algorithm takes to run. It's how we compare the efficiency of different approaches to a problem.

With big O notation we express the runtime in terms of how quickly it grows relative to the input, as the input gets arbitrarily large.

Aspects to look out for while studying Big O notation:

1. how quickly the runtime grows—Some external factors affect the time it takes for a function to run: the speed of the processor, what else the computer is running, etc. So it's hard to make strong statements about the exact runtime of an algorithm. Instead we use big O notation to express how quickly its runtime grows.
2. relative to the input—Since we're not looking at an exact number, we need to phrase our runtime growth in terms of something. We use the size of the input. So we can say things like the runtime grows "on the order of the size of the input" O(n) or "on the order of the square of the size of the input" O(n^2).
3. as the input gets arbitrarily large—Our algorithm may have steps that seem expensive when n is small but are eclipsed eventually by other steps as n gets huge. For big O analysis, we care most about the stuff that grows fastest as the input grows, because everything else is quickly eclipsed as n gets very large. If you know what an asymptote is, you might see why "big O analysis" is sometimes called "asymptotic analysis."

SOURCE: https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity

ALGORITHMS IN THIS REPORT

In this report I will analyze and compare the following searching algorithms:

1. Straight Sequential Search
2. Ordered Sequential Search
3. Indexed Sequential Search
4. Straight Sequential Search with First Element Swap
5. Straight Sequential Search with Previous Element Swap
6. Binary Search

HYPOTHESIS

I think Binary Search will have the best average running time, followed by Indexed Sequential Search. A Binary Search has a complexity of O(n log·n) and the Indexed Sequential Search has a complexity of O($k \cdot n^{1/(k+1)}$).
For this case study's base case **[10^2 | k = 2 | n = 100]**:
Binary Search ➔ approx. 460 steps each search → there are 10^3 search keys → 460,000 steps in total.
Indexed Sequential Search ➔ approx. 10 steps each search → 10,000 steps in total.

While Indexed Sequential Search does better for the smallest case, I am hypothesizing that given its growth is exponential, the average number of steps for Binary Search will end up being less – meaning Binary Search will do better in the longer term.

STUDYING INDIVIDUAL ARRAY SIZES:

ARRAY OF SIZE 10^2

INDEXED SEQUENTIAL SEARCH	50732.4
ORDERED SEQUENTIAL SEARCH	100017.36
PREVIOUS SWAP	100033.03
FIRST SWAP	100036.59
STRAIGHT SEQUENTIAL SEARCH	100490.86
BINARY SEARCH	376193.05

Number of Runs = **100**
Average Number of Matches found = **24**

ARRAY OF SIZE 10^4

BINARY SEARCH	1,468,919.00
INDEXED SEQUENTIAL SEARCH	2,500,750.00
ORDERED SEQUENTIAL SEARCH	10,000,000.00
FIRST SWAP	10,000,000.10
PREVIOUS SWAP	10,000,000.10
STRAIGHT SEQUENTIAL SEARCH	10,000,549.91

Number of Runs = **100**
Average Number of Matches found = **17**

ARRAY OF SIZE 10^6

BINARY SEARCH	3,187,792.37
INDEXED SEQUENTIAL SEARCH	166,579,666.49
ORDERED SEQUENTIAL SEARCH	999,211,000.29
PREVIOUS SWAP	999,211,001.35
FIRST SWAP	999,211,001.39
STRAIGHT SEQUENTIAL SEARCH	999,628,190.51

Number of Runs = **100**
Average Number of Matches found = **41**

ARRAY OF SIZE 10^8

Binary Search	2,596,394.11
Ordered Sequential Search	182,802,274.45
Previous Element Swap	182,802,313.18
First Element Swap	182,802,431.68
Indexed Sequential Search	545,756,080.24
Straight Sequential Search	897,411,718.16

Number of Runs = **45**
Average Number of Matches found = **96**

STUDYING OVERALL ALGORITHM PERFORMANCES

BINARY SEARCH	1,907,324.63
INDEXED SEQUENTIAL SEARCH	42,282,787.22
ORDERED SEQUENTIAL SEARCH	298,028,323.02
PREVIOUS SWAP	298,028,336.92
FIRST SWAP	298,028,367.44
STRAIGHT SEQUENTIAL SEARCH	476,785,237.36

ANALYSIS VS HYPOTHESIS

The overall average algorithm performance confirms my hypothesis, Binary Search showed a remarkably better performance compared to all other algorithms. Compared to the next fastest algorithm – Indexed Sequential Search – Binary Search was about 21 times faster. Indexed Sequential Search was approximately 8 times faster than the next 3 algorithms – Ordered, Previous Swap, and First Swap. Not surprisingly Straight Sequential Search showed the worst performance.

LOOKING AT THE CODE

PRE-CODE RESEARCH

After my first attempt to create an array of size 10^8, I realized that I had to optimize the random number generator I was using to minimize the time it took to create the arrays. The solution I found was to use a combination of two classes found in C++ "random.h".

I used the *mt19937* to generate the numbers. Also from "random.h", I used *random_device* to generate a random seed which *mt19937* would use to generate the numbers.

mt19937

mt19937 is random number generator based on the Mersenne Twister 19937 generator: a pseudo-random number generator based on equations that recursively defines a sequence or multidimensional array of values. **[source: https://en.wikipedia.org/wiki/Mersenne_Twister]**

random_device

random_device is a true random number generator – a random generator that uses the computer hardware to generate the random numbers.
[source: https://en.wikipedia.org/wiki/Hardware_random_number_generator]

A combination of these two classes guaranteed a true random number generation each time.

```
//Random Number Generator Engine
random_device r;
seed_seq seed{ r() }; //Returns seed to the mt19937 engine below
mt19937 eng(seed);
uniform_int_distribution<> dist{ 0, sizeOfArray };
//0 and sizeOfArray are limits for the random number generator
```

GENERATING ARRAYS

Having already defined the random number generator, generating the array was as simple as going through a for loop.

```
	for (int i = 0; i < sizeOfArray; i++) {
		_toSearchArray[i] = dist(eng);
		// See above for dist(eng) : it returns a new random number
	}
```

STRAIGHT SEQUENTIAL SEARCH

A straight sequential search will have a complexity O(n^2) because it is a for loop nested within another for loop – one for the search keys and another for the array through which we will search.

```
// _array is the array to search through
// _arraySize is the size of the array to search through
// _keys is the array containing search keys
void straightSequence(int *_array, int _arraySize, int *_keys) {
	outputFile <<  "Running Straight Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	for (int i = 0; i < pow(10,3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_stepCount++;
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _stepCount << "\t" << _matchCount << endl;
}
```

MORE ON NEXT PAGE >>

QUICK SORT

Before we can see implement Ordered Sequential Search – we will need to implement a sorting algorithm. I decided to use the quick sort algorithm - being the fastest sorting algorithm that I am familiar with and that I can implement. Complexity: O(n log(n))

```
void quickSort(int arr[], int left, int right) {
	int i = left, j = right;
	int tmp;
	int pivot = arr[(left + right) / 2];
	while (i <= j) {
		while (arr[i] < pivot)
			i++;
		while (arr[j] > pivot)
			j--;
		if (i <= j) {
			tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	};
	if (left < j)
		quickSort(arr, left, j);
	if (i < right)
		quickSort(arr, i, right);
}
```

ORDERED SEQUENTIAL SEARCH

After having implemented the quicksort, Ordered Sequential Search becomes as simple as just Straight Sequential Search on the sorted array. $O(n \log(n)) + O(n^2)$

```
void orderedSequence(int *_array, int _arraySize, int *_keys) {
	outputFile << "Running Ordered Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	quickSort(_array, 0, _arraySize-1);
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _stepCount << "\t" << _matchCount << endl;
}
```

MORE ON NEXT PAGE >>

INDEXED SEQUENTIAL SEARCH

Indexed Sequential Search works by jumping through the array by **k** each time, since it is going through a sorted array it checks to see if the cursor has passed the value of the key. If so, the cursor will iterate backwards till the last jump spot. Complexity: $O(k \cdot n^{1/(k+1)})$.

```
void indexedSequence(int *_array, int _arraySize, int _exp, int *_keys) {
	outputFile << "Running Indexed Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	quickSort(_array, 0, _arraySize - 1);
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j+=log10(_arraySize), _stepCount++)
{
			if (_array[j] > _keys[i]){
				for (int k = j-1; k > j - log10(_arraySize); k--) {
					if (_array[k] == _keys[i]) {
						_matchCount++;
						break;
					}
					else
					{
						break;
					}
				}
			}
			else if (_array[j] == _keys[i]) {
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _stepCount << "\t" << _matchCount << endl;
}
```

MORE ON NEXT PAGE >>

SEQUENTIAL SEARCH WITH FIRST OR PREVIOUS ELEMENT SWAP

Sequential search with first **OR** previous element swap both have complexities of $O(n^2 + 3)$. Their implementation is very straight forward and almost perfectly matches the Straight Sequential Search.

```
//0 is replaced with [j-1] for the PREVIOUS ELEMENT SWAP

void firstSequence(int *_array, int _arraySize, int *_keys) {
	outputFile <<"Running Straight Sequential Search with First Element
Swap..." << "\t"; ;
	int _stepCount = 0, _matchCount = 0;
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_matchCount++;
				_stepCount++;
				if (j != 0) {
					int temp = _array[0];
					_array[0] = _array[j];
					_array[j] = temp;
					_stepCount += 3;
				}
				break; //this
			}
		}
	}
	outputFile << _stepCount << "\t" << _matchCount << endl;
}
```

MORE ON NEXT PAGE >>

BINARY SEARCH

Binary Search has a complexity of O (n log(n)).

PROCEDURE: Given an array A of n elements with values or records A0 ... An−1, sorted such that A0 ≤ ... ≤ An−1, and target value T, the following subroutine uses binary search to find the index of T in A.[6]

1. Set L to 0 and R to n − 1.
2. If L > R, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor (the largest previous integer) of (L + R) / 2.
4. If Am < T, set L to m + 1 and go to step 2.
5. If Am > T, set R to m – 1 and go to step 2.
6. Now Am = T, the search is done; return m.

This iterative procedure keeps track of the search boundaries via two variables. Some implementations may place the comparison for equality at the end of the algorithm, resulting in a faster comparison loop but costing one more iteration on average.

```
int _binaryStepCount = 0;
int binarySearch(int arr[], int value, int left, int right) {
	while (left <= right) {
		int middle = (left + right) / 2;
		if (arr[middle] == value) {
			_binaryStepCount++;
			return middle;
		}
		else if (arr[middle] > value) {
			_binaryStepCount++;
			right = middle - 1;
		}
		else {
			left = middle + 1;
			_binaryStepCount++;
		}
	}
	return -1;
}

void binary(int *_array, int _arraySize, int *_keys) {
	outputFile << "Running Binary Search..." << "\t";
	int _matchCount = 0;
	quickSort(_array, 0, _arraySize - 1);
	for (int i = 0; i < pow(10, 3); i++, _binaryStepCount++) {
		if (-1 != binarySearch(_array, _keys[i], 0, _arraySize)) {
			_matchCount++;
		}
	}
	outputFile << _stepCount << "\t" << _matchCount << endl;
}
```

APPENDIX: CODE [MAIN.CPP]

```
#include <iostream>
#include <fstream>
#include <random>
#include <climits>
#include <stdlib.h>
#include <algorithm>
#include <string>

using namespace std;
ofstream outputFile;

void straightSequence(int *_array, int _arraySize, int *_keys) {
	outputFile <<  "Running Straight Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	for (int i = 0; i < pow(10,3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_stepCount++;
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _matchCount << endl;
}

void quickSort(int arr[], int left, int right) {
	int i = left, j = right;
	int tmp;
	int pivot = arr[(left + right) / 2];
	while (i <= j) {
		while (arr[i] < pivot)
			i++;
		while (arr[j] > pivot)
			j--;
		if (i <= j) {
			tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	};
	if (left < j)
		quickSort(arr, left, j);
	if (i < right)
		quickSort(arr, i, right);
}
```

```
void orderedSequence(int *_array, int _arraySize, int *_keys) {
	outputFile << "Running Ordered Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	quickSort(_array, 0, _arraySize-1);
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _matchCount << endl;
}

void indexedSequence(int *_array, int _arraySize, int _exp, int *_keys) {
	outputFile << "Running Indexed Sequential Search..." << "\t";;
	int _stepCount = 0, _matchCount = 0;
	quickSort(_array, 0, _arraySize - 1);
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j+=log10(_arraySize), _stepCount++)
{
			if (_array[j] > _keys[i]){
				for (int k = j-1; k > j - log10(_arraySize); k--) {
					if (_array[k] == _keys[i]) {
						_matchCount++;
						break;
					}
					else
					{
						break;
					}
				}
			}
			else if (_array[j] == _keys[i]) {
				_matchCount++;
				break;
			}
		}
	}
	outputFile << _matchCount << endl;
}

void firstSequence(int *_array, int _arraySize, int *_keys) {
	outputFile <<"Running Straight Sequential Search with First Element
Swap..." << "\t"; ;
	int _stepCount = 0, _matchCount = 0;
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_matchCount++;
				_stepCount++;
				if (j != 0) {
					int temp = _array[0];
					_array[0] = _array[j];
					_array[j] = temp;
					_stepCount++;
				}
				break; //this
```

```
}
		}
	}
	outputFile << _matchCount << endl;
}

void swapSequence(int *_array, int _arraySize, int *_keys) {
	outputFile << "Running Straight Sequential Search with Previous Element
Swap..." << "\t";
	int _stepCount = 0, _matchCount = 0;
	for (int i = 0; i < pow(10, 3); i++, _stepCount++) {
		for (int j = 0; j < _arraySize; j++, _stepCount++) {
			if (_keys[i] == _array[j]) {
				_matchCount++;
				_stepCount++;
				if (j != 0) {
					int temp = _array[j-1];
					_array[j-1] = _array[j];
					_array[j] = temp;
					_stepCount++;
				}
				break; //this
			}
		}
	}
	outputFile << _matchCount << endl;
}

int _binaryStepCount = 0;
int binarySearch(int arr[], int value, int left, int right) {
	while (left <= right) {
		int middle = (left + right) / 2;
		if (arr[middle] == value) {
			_binaryStepCount++;
			return middle;
		}
		else if (arr[middle] > value) {
			_binaryStepCount++;
			right = middle - 1;
		}
		else {
			left = middle + 1;
			_binaryStepCount++;
		}
	}
	return -1;
}

void binary(int *_array, int _arraySize, int *_keys) {
	outputFile << "Running Binary Search..." << "\t";
	int _matchCount = 0;
	quickSort(_array, 0, _arraySize - 1);
	for (int i = 0; i < pow(10, 3); i++, _binaryStepCount++) {
		if (-1 != binarySearch(_array, _keys[i], 0, _arraySize)) {
			_matchCount++;
		}
	}
	outputFile <<_matchCount << endl;
}
```

```
void cycle(int _userSize, int _run) {
	//Random Number Generator Engine for the Search Array
	random_device r;
	seed_seq seed{ r() };
	mt19937 eng(seed);

	//Random Number Generator Engine for the Search Array
	random_device r2;
	seed_seq seed2{ r2() };
	mt19937 eng2(seed2);
	int *_toSearchArray, _pow10size, *_searchKeys;

	outputFile << "\nRun " << _run << "\n";

	if (_userSize >= 6) {
		_pow10size = INT_MAX / 2;
	}
	else
	{
		_pow10size = pow(10, _userSize * 2);
	}
	_userSize = pow(10, _userSize);
	_toSearchArray = new (nothrow) int[_userSize];

	//Makes sure there was no problem in allocating the memory for the array
	if (_toSearchArray == nullptr) {
		cout << "Failed to allocate memory." << endl;
	}

	//Random integers are inserted into the _toSearchArray
	uniform_int_distribution<> dist{ 0, _pow10size };
	for (int i = 0; i < _userSize; i++) {
		_toSearchArray[i] = dist(eng);
	}

	outputFile << "Array of size : " << _userSize << "\n";

	int searchSize = pow(10, 3);

	//Generating integers to search for
	std::uniform_int_distribution<> searchDist{ 0 , searchSize };
	_searchKeys = new int[searchSize];
	for (int i = 0; i < 1000; i++) {
		_searchKeys[i] = searchDist(eng2);
	}

	straightSequence(_toSearchArray, _userSize, _searchKeys);
	orderedSequence(_toSearchArray, _userSize, _searchKeys);
	indexedSequence(_toSearchArray, _userSize, log10(_userSize), _searchKeys);
	firstSequence(_toSearchArray, _userSize, _searchKeys);
	swapSequence(_toSearchArray, _userSize, _searchKeys);
	binary(_toSearchArray, _userSize, _searchKeys);
}
```

```
int main() {

	int values[] = {4,6};

	for (int i = 0; i < 2; i++) {
		string fileName = to_string(values[i]) + ".txt";
		outputFile.open(fileName);
		for (int j = 0; j < 1; j++) {
			cycle(values[i], j);
		}
		outputFile.close();
	}
	return 0;
}
```