

# CS201

# Fundamental Structures of Computer Science 1

Bilkent University

Fall 2024/2025

Homework Assignment 2

16.11.2024, Saturday

Section 3

Ferit Bilgi

22102554

**Device Specifications**

Macbook Pro (14 inch, early 2023), Apple

**CPU:**

Apple M2. Pro chip

Total Cores: 10

Total Threads: 10

Cache: 36 MB

Max Turbo Frequency: 3.7 GHZ

Processor Base Frequency: 3.4 GHZ

**Memory:**

16GB unified memory

**Operating System:**

macOS Sequoia

version 15.0.1

**Development Environment**

IDE: CLion 2024.2.1, JetBrains

Compiler: C++17

# Algorithm Analysis

**Linear Search (Iterative)**

Linear search iteratively goes through each element of the array until it finds the desired key or reaches the end. Its time complexity is O(N) where N is the size of the array, so its performance linearly depends on the input size. It is good in cases where the key is near the beginning of the array and only a few comparisons are needed. In the worst case—that is, when the key is at the end or not in the array—every element has to be checked, leading to

N comparisons. Where it is simple and efficient for small data sets, it is inefficient with large arrays because it cannot take advantage of any structural properties of the data.

**Linear Search (Recursive)**

The recursive version of linear search is implemented similarly to the iterative approach with the main difference being that the step is implemented using a recursive call. Even though its time complexity is O(N), in practice, it's less efficient because of the overhead caused by the recursive function calls. This extra overhead in this function can also cause a stack overflow for very large input sizes, thus making it less robust than the iterative version. It has optimal performance when the key is at the beginning of the array, and worst performance when the key is at the end or, in fact, not there. While it's utterly impractical for large sets of data, the function is an interesting example of using recursion and is helpful in understanding the concept.

**Binary Search**

Thus, binary search exploits the fact that a given array is sorted to reduce the time complexity to O(logN). It achieves this by dividing the search space in half every time, thus fastening the possible location of the key in the array. Therefore, binary search is far faster for large data sets than linear search. Its performance in all cases remains because it depends on logarithmic halving, regardless of the key's position. It is, however, limited to sorted arrays and hence unsuitable for unsorted data unless the latter has been sorted. In such cases where the data is already sorted or subjected to frequent searches, binary search is the most efficient method.

**Jump Search**

Jump search combines the advantages of linear and binary search, since it skips fixed-size blocks of elements in order to narrow down the search range before performing a linear scan within the block. Its time complexity is O(sqrt(N)), which makes it more efficient than linear search and slower than binary search for large arrays. It works best when the key is located at or near the start of a block, as fewer comparisons need to be made in the block. For small datasets, the overhead of skipping blocks can make it less efficient than simple linear search. In general, jump search is a good practical alternative to the binary search, especially where the latter is not applicable to sorted data.

**Random Seach**

Random search checks out the elements in a random order until it finds the desired key. The time complexity theoretically is O(N), but due to randomness, it is inherently unpredictable and usually inefficient in practice. With small arrays, random search may accidentally perform well if the key is randomly checked early. But in the worst case—that is, when the key happens to be the last to be checked or is not present—every element must be searched, just like linear search. Because of its inconsistency and lack of efficiency, random search is not really a practical algorithm but could be used for experimental purposes or as an example of how randomness can be brought into algorithm design for educational purposes.

# Execution Time Results

Linear (ITERATIVE)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | SCENARIO 1 | SCENARIO 2 | SCENARIO 3 | SCENARIO 4 |
| 1000 | 0.004 MS | 0.001 MS | 0.002 MS | 0.002 MS |
| 10000 | 0.001 MS | 0.011 MS | 0.018 MS | 0.018 MS |
| 100000 | 0.004 MS | 0.079 MS | 0.163 MS | 0.159 MS |

Linear (RECURSIVE)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | SCENARIO 1 | SCENARIO 2 | SCENARIO 3 | SCENARIO 4 |
| 1000 | 0.004 MS | 0.006 MS | 0.012 MS | 0.011 MS |
| 10000 | 0.001 MS | 0.075 MS | 0.144 MS | 0.116 MS |
| 100000 | 0.001 MS | 0.764 MS | 1.242 MS | 1.078 MS |

Binary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | SCENARIO 1 | SCENARIO 2 | SCENARIO 3 | SCENARIO 4 |
| 1000 | 0.00071429 MS | 0.00042857 MS | 0.00042857 MS | 0.00057143 MS |
| 10000 | 0.00071429 MS | 0.00071429 MS | 0.001571429MS | 0.0071429 MS |
| 100000 | 0.001 MS | 0.001 MS | 0.001 MS | 0.002 MS |

Jump Search (includes averages for each scenario)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | SCENARIO 1 | SCENARIO 2 | SCENARIO 3 | SCENARIO 4 |
| 1000 | 0.00042857 MS | 0.001 MS | 0.00042857 MS | 0.00057143 MS |
| 10000 | 0.00014286 MS | 0.001 MS | 0.001MS | 0.001142857MS |
| 100000 | 0.00057143 MS | 0.00085714 MS | 0.00057143 MS | 0.00042857 MS |

Random Search

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | SCENARIO 1 | SCENARIO 2 | SCENARIO 3 | SCENARIO 4 |
| 1000 | 0.013 MS | 0.015 MS | 0.015 MS | 0.014 MS |
| 10000 | 0.123 MS | 0.132 MS | 0.143 MS | 0.144 MS |
| 100000 | 1.192 MS | 1.249 MS | 1.258 MS | 1.274 MS |

# Graphs