CS404 Project Search/Sort Algorithms Efficiency

Alex G, Ethan N, Victor O
4/28/2025

Agenda

- Overview
- Objectives
- Methodology
- Functions
- Key Findings
- Examples
- Conclusion
- Sources

Overview

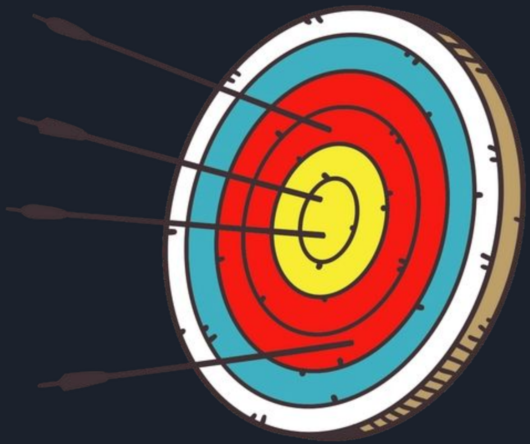
- Search Algorithms help find the data point the user wants in an efficient manner, while sorting algorithms organize the data from the smallest to biggest value
- Measure efficiency through time complexity

- Linear search
  - WC: O(n)
  - AC: O(n)
  - BC: O(1)
- Binary search
  - WC: O(log n)
  - AC: O(log n)
  - BC: O(1)

- Insertion Sort
  - WC: O(n^2)
  - AC: O(n^2)
  - BC: O(1)
- Merge Sort
  - WC: O(n log n)
  - AC: O(n log n)
  - BC: O(1)
- Radix Sort
  - O(nk) where k is the number of digits in the highest value

Objectives

- Implement/Compare the time complexity of different searching algorithms (linear and binary search)
- Implement/Compare the time complexity of different sorting algorithms (insertion, merge, and radix sort)
- Have a easy to use Program to test and visualize the results of the Search and Sorting algorithms



Methodology

1. Chose search and sort algorithms based on their simplicity and efficiency
2. Implemented in code
3. Implement test case
4. Measured and analyzed times
5. Allow user to choose what data to visualize

Modules used:
Random
Time
matplotlib

```
Analyzing search and sort algorithm performance...

Input Size |   Linear (s) |   Binary (s) |   Insertion (s) |    Merge (s) |    Radix (s)
------------------------------------------------------------------------------------
        10 |   0.00000281 |   0.00000273 |      0.00000948 |   0.00002422 |   0.00002232
       100 |   0.00000719 |   0.00000269 |      0.00041083 |   0.00019124 |   0.00010040
      1000 |   0.00004434 |   0.00000294 |      0.08256522 |   0.00428464 |   0.00233272
      5000 |   0.00022104 |   0.00000515 |      1.92360680 |   0.06823886 |   0.01211916
     10000 |   0.00107449 |   0.00000809 |      8.08981261 |   0.08394549 |   0.07689826
     20000 |   0.00129382 |   0.00000829 |     17.02912430 |   0.09438843 |   0.05237424

Time Complexity Discussion:
- Linear Search: O(n)
- Binary Search: O(log n)
- Insertion Sort: O(n^2) worst case, O(n) best case
- Merge Sort: O(n log n) in all cases
- Radix Sort: O(nk), where k is number of digits
```

```
Choose which graph to display:
1. Search Algorithms Performance
2. Sorting Algorithms Performance
3. Combined Search and Sort (Log Scale)
4. Exit
Enter your choice (1-4):
```

Functions

```python
# Linear and Binary search algorithms
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

```python
def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i-1]

    i = n - 1
    while i >= 0:
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
        i -= 1

    for i in range(n):
        arr[i] = output[i]
```

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr
```

```python
def radix_sort(arr):
    if len(arr) == 0:
        return arr
    max1 = max(arr)
    exp = 1
    while max1 // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10
    return arr
```

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr
```

Functions continued

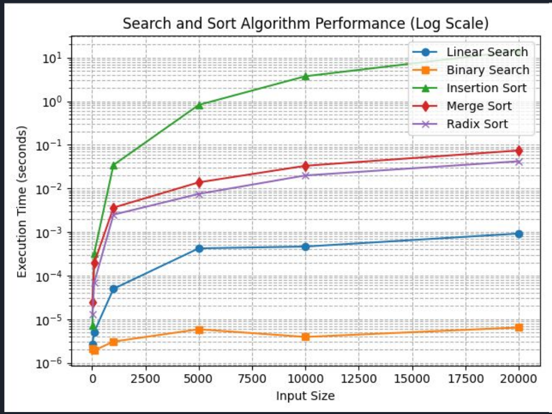
```python
for size in input_sizes:
    test_data = list(range(size))  # Sorted data for searches
    random_data = [random.randint(0, size) for _ in range(size)]  # Random data for sorting

    target = size - 1  # Worst case for linear search

    # Linear Search
    start_time = time.perf_counter()
    linear_search(test_data, target)
    linear_time = time.perf_counter() - start_time
    linear_times.append(linear_time)

    # Binary Search
    start_time = time.perf_counter()
    binary_search(test_data, target)
    binary_time = time.perf_counter() - start_time
    binary_times.append(binary_time)

    # Insertion Sort
    insertion_data = random_data.copy()
    start_time = time.perf_counter()
    insertion_sort(insertion_data)
    insertion_time = time.perf_counter() - start_time
    insertion_times.append(insertion_time)

    # Merge Sort
    merge_data = random_data.copy()
    start_time = time.perf_counter()
    merge_sort(merge_data)
    merge_time = time.perf_counter() - start_time
    merge_times.append(merge_time)

    # Radix Sort
    radix_data = random_data.copy()
    start_time = time.perf_counter()
    radix_sort(radix_data)
    radix_time = time.perf_counter() - start_time
    radix_times.append(radix_time)

    print(f"{size:>10} | {linear_time:>12.8f} | {binary_time:>12.8f} | {insertion_time:>15.8f} |
{merge_time:>12.8f} | {radix_time:>12.8f}")
```

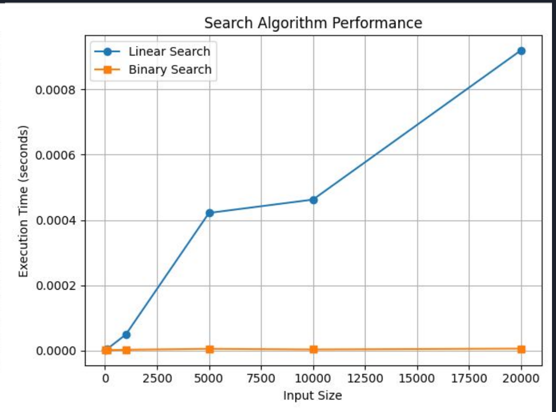
```python
def plot_search_algorithms(input_sizes, linear_times, binary_times):
  plt.figure()
  plt.plot(input_sizes, linear_times, label='Linear Search', marker='o')
  plt.plot(input_sizes, binary_times, label='Binary Search', marker='s')
  plt.xlabel('Input Size')
  plt.ylabel('Execution Time (seconds)')
  plt.title('Search Algorithm Performance')
  plt.legend()
  plt.grid(True)
  plt.tight_layout()
  plt.show()

def plot_sort_algorithms(input_sizes, insertion_times, merge_times, radix_times):
  plt.figure()
  plt.plot(input_sizes, insertion_times, label='Insertion Sort', marker='o')
  plt.plot(input_sizes, merge_times, label='Merge Sort', marker='s')
  plt.plot(input_sizes, radix_times, label='Radix Sort', marker='^')
  plt.xlabel('Input Size')
  plt.ylabel('Execution Time (seconds)')
  plt.title('Sorting Algorithm Performance')
  plt.legend()
  plt.grid(True)
  plt.tight_layout()
  plt.show()
```

```python
def plot_combined_algorithms(input_sizes, linear_times, binary_times, insertion_times, merge_times,
radix_times):
  plt.figure()
  plt.plot(input_sizes, linear_times, label='Linear Search', marker='o')
  plt.plot(input_sizes, binary_times, label='Binary Search', marker='s')
  plt.plot(input_sizes, insertion_times, label='Insertion Sort', marker='^')
  plt.plot(input_sizes, merge_times, label='Merge Sort', marker='d')
  plt.plot(input_sizes, radix_times, label='Radix Sort', marker='x')
  plt.xlabel('Input Size')
  plt.ylabel('Execution Time (seconds)')
  plt.title('Search and Sort Algorithm Performance (Log Scale)')
  plt.yscale('log')
  plt.legend()
  plt.grid(True, which="both", ls="--")
  plt.tight_layout()
  plt.show()
```

Key Findings

- As each data point deviated from index 0, linear search became less efficient in searching the number, whereas binary search stayed relatively constant in the time it took to search for a number
- For smaller input sizes, the difference between the time for binary and linear search is negligible, but increases for larger input sizes as it took more time for linear search.
- For speed of sorting algorithms, Radix > Merge > Insertion as input size increases
- These results are consistent with their time complexities.





Examples

- Comparing and analyzing time complexities for search algorithms is important as it helps to get users the results they want as efficiently as possible
  - Googling which AI LLM is best for coding
  - Finding past emails
  - Ordering from Amazon

While saving fractions of a second might not matter when running small programs, for large companies, even small gains in efficiency can significantly impact overall performance, scalability, and cost savings.

Conclusion

- For smaller input sizes, linear and binary search perform similarly
- For larger input sizes, binary search takes significantly less time to search for a data point than linear search does.
    - This is expected as the worst/average case of linear search is O(n) and the worst/average case of binary search is O(log n)
- As input size increases, Radix Sort is the quickest, followed by Merge, followed by Insertion
    - This is expected because insertion sort has the time complexity of n^2 for average case, merge sort has a time complexity of logn

Questions

References

Algorithms for Efficient File Searching: Mastering the Art of Quick Data Retrieval. AlgoCademy Blog. (n.d.). https://algocademy.com/blog/algorithms-for-efficient-file-searching-mastering-the-art-of-quick-data-retrieval/

GeeksforGeeks. (2023, December 19). *Linear Search vs Binary Search*. GeeksforGeeks. https://www.geeksforgeeks.org/linear-search-vs-binary-search/