Sorts
Understanding Array Sorting Algorithms, Efficiency, Comparison and Implementation

INSAlgo
March 19, 2024

Onyr (Florian RASCOUSSIER)

INSA Lyon & IMT Atlantique

florian.rascoussier@insa-lyon.fr

github.com/onyr

Runtime and Memory Complexity: Basics

- **Runtime Complexity**
  - Time an algorithm takes relative to input length.
- **Memory Complexity**
  - Memory needed by an algorithm relative to input size.
- Both are crucial for:
  - Comparing algorithm efficiency.
  - Choosing the right algorithm for the job.

Measuring Program Execution Time: Challenges

Types of Time Measurements

- **Time Measurement Issues**
  - Real Time: Wall-clock time for program execution.
  - User Time: CPU time for executing user program code.
    - Excludes system operations.
    - Reflects direct program execution time.
  - System Time[1]: CPU time for system operations for the program.
    - File operations, I/O tasks.
    - Essential for resource-intensive operations.
- Variability in measurements due to:
  - System load, resources, hardware.
  - Inconsistencies across environments.

[1]See https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1

Measuring Program Execution Time: Theoretical Approach

Abstracting Time Measurements

- **Theoretical Approach**
  - Approximate with **input size** (n) and **operation count**.
  - $n \in \mathbb{N}^*$: Number of loops or iterations – main driver of complexity.
  - $k \in \mathbb{N}^*$: Parameters affecting complexity, aside from input size.
    - Here intended as range of the non-negative key values
  - Focus on growth trends rather than exact times.
- **Conclusion:**
  - Theoretical focus helps identify scalability issues.
  - Prioritizes relative efficiency over absolute timing.

Understanding Big O Notation

- **Big O Notation**: Describes the upper bound of complexity.
  - Focuses on worst-case scenario.
  - Ignores constant factors and lower order terms.
- **Basic Rules**
  - *Linear Terms*: $\mathcal{O}(\alpha n + \beta) = \mathcal{O}(n)$.
    - Constants α, β don't affect growth rate.
  - *Sum Rule*: $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n), g(n)))$.
  - *Product Rule*: $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$.
- **Implications**
  - Simplifies comparing algorithms.
  - Emphasizes dominant factors affecting growth.
- **Example**
  - $\mathcal{O}(3n^2 + 10n + 100) = \mathcal{O}(n^2)$.
    - n^2 term dominates as n grows.

Classic Sorting Algorithms and Their Complexities

Algo.	Best	Average	Worst	Mem.
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$

More Algorithms and Their Complexities[2]

Algorithm	Best	Average	Worst	Memory
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Timsort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Tree Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n(\log n)^2)$	$O(n(\log n)^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

[2] See https://www.bigocheatsheet.com/

Selection Sort Algorithm

Simplicity in Action

- **How It Works**
  - Iteratively selects the smallest element from the unsorted portion and swaps it with the element at the current position.
  - Continues until the entire array is sorted.

```python
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Insertion Sort Algorithm

Building the Sorted Array

- **How It Works**
  - Builds the sorted array one element at a time.
  - Iterates through the array, shifting elements to the right to make space for the current element.

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

Bubble Sort Algorithm Bubbling Up the Largest Element

- **How It Works**
  - Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
  - Continues until no more swaps are needed.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Quicksort Algorithm

Divide and Conquer

- **How It Works**
  - Employs a divide-and-conquer strategy to sort the array.
  - Selects a 'pivot' element and partitions the array into sub-arrays of elements less than and greater than the pivot.
  - Recursively applies the same strategy to the sub-arrays.
  - Highly efficient for large datasets, with average time complexity of O(n log n).

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

Merge Sort Algorithm

- **How It Works**
  - Another divide-and-conquer algorithm that divides the array into halves, sorts each half, and then merges the sorted halves together.
  - Begins with the division of the array into smallest manageable units, then merges units in a sorted manner to form larger sorted sections until the whole array is merged back together.
  - Consistently performs with a time complexity of O(n log n), making it highly efficient for large data sets.
- **Key Features**
  - Predictable performance.
  - Stable sorting algorithm.
  - Popular. Default sorting algorithm for many programming languages, but tends to be replaced by Timsort in practice.

Merge Sort Algorithm

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

Thank You for Your Attention!

Further Resources

- **Useful Links**
  - Big O Cheat Sheet: https://www.bigocheatsheet.com/
    - A handy reference for complexity of common data structures and algorithms.
  - Sorting Algorithms with Animations: https://www.toptal.com/developers/sorting-algorithms/bubble-sort
    - Explore how different sorting algorithms work with interactive animations.
- **Contact & Feedback**
  - My GitHub: https://github.com/0nyr
  - This presentation: https://github.com/0nyr/sorting_algorithms

Once again, thank you and have a great day!