Algorithm Design and Optimization

(ICT 341-2)

(Assignment 01)

R S C Wijesekara
UWU/ICT/22/012
Bachelor of Information and Communication Technology Degree Program
Department of Information and Communication Technology
Faculty of Technological Studies
Uva Wellassa University of Sri Lanka

(a)

Definition of an Algorithm

An algorithm is a finite, well-defined sequence of instructions or steps designed to solve a specific problem or perform a particular task. It takes input, processes it through a series of unambiguous operations, and produces an output.

Key characteristics of an algorithm:

- **Finiteness**: It must terminate after a finite number of steps
- **Definiteness**: Each step must be precisely defined with no ambiguity
- **Input**: It has zero or more inputs
- **Output**: It produces at least one output
- **Effectiveness**: Each operation must be basic enough to be performed exactly and in finite time

What Makes an Algorithm Efficient

Efficiency in algorithms is typically measured along two primary dimensions:

Time Complexity: How the algorithm's execution time grows relative to input size. An efficient algorithm minimizes the number of operations needed. For example, binary search (O(log n)) is more time-efficient than linear search (O(n)) for sorted data.

Space Complexity: How much memory the algorithm requires relative to input size. Efficient algorithms use memory judiciously, avoiding unnecessary storage.

Additional factors affecting efficiency include:

- **Scalability**: How well the algorithm performs as input size increases dramatically
- **Optimality**: Whether the algorithm achieves the best possible time/space tradeoff for the problem
- **Practical performance**: Real-world factors like cache usage, constant factors hidden in big-O notation, and hardware-specific optimizations
- **Problem-appropriate complexity**: An algorithm that matches or comes close to the theoretical lower bound for solving that particular problem

The most efficient algorithm balances these factors based on the specific requirements and constraints of the application, recognizing that sometimes optimizing for time means using more space, and vice versa.

(b)

Four Factors That Affect Algorithm Running Time

1. Input Size (n)

The size of the input is typically the most significant factor affecting running time. As the number of elements to process increases, the execution time grows according to the algorithm's time complexity. For example, sorting 1,000 elements takes considerably less time than sorting 1,000,000 elements. The relationship between input size and running time is expressed through big-O notation ($O(n)$, $O(n^2)$, $O(\log n)$, etc.). An algorithm with $O(n^2)$ complexity will see its running time quadruple when input size doubles.

2. Input Characteristics and Organization

The specific nature and arrangement of input data significantly impacts performance. Some algorithms are sensitive to whether data is already sorted, partially sorted, or randomly distributed. For instance, Quick Sort performs optimally with randomly distributed data but degrades to $O(n^2)$ with already sorted input if a poor pivot selection strategy is used. Similarly, searching algorithms perform differently depending on whether the target element is at the beginning, middle, or end of a dataset. Best case, average case, and worst case scenarios often differ dramatically based on input characteristics.

3. Hardware and System Resources

The physical computing environment directly affects execution time. Faster processors execute instructions more quickly, while larger amounts of RAM allow algorithms to avoid slower disk based operations. Cache memory size and efficiency influence how quickly data can be accessed, and CPU architecture (number of cores, instruction set) determines parallel processing capabilities. Additionally, system load from other running processes competes for resources, and the efficiency of the compiler or interpreter translating code into machine instructions affects the final execution speed.

4. Algorithm Implementation Quality

How an algorithm is coded and the programming language used substantially impact running time. Efficient implementations minimize unnecessary operations, avoid redundant calculations, and use appropriate data structures. Language choice matters compiled languages like C++ generally execute faster than interpreted languages like Python for the same algorithm. Programming practices such as avoiding nested loops when possible, using efficient built in functions, minimizing memory allocations, and optimizing recursion versus iteration all contribute to the actual running time beyond the theoretical complexity analysis.

(c)

Asymptotic Notation Definitions

Big-O Notation (O) - Upper Bound

Big-O notation describes the worst-case or upper bound of an algorithm's growth rate. It represents the maximum time or space an algorithm could require as input size approaches infinity. When we say an algorithm is O(f(n)), we mean that its running time grows no faster than f(n) for sufficiently large n, ignoring constant factors and lower-order terms. For example, $O(n^2)$ means the algorithm's running time will not exceed a constant multiple of n^2 for large inputs. Big-O provides a guarantee that the algorithm will not perform worse than the specified bound, making it useful for understanding worst-case performance and ensuring an algorithm meets performance requirements.

Omega Notation (Ω) - Lower Bound

Omega notation describes the best-case or lower bound of an algorithm's growth rate. It represents the minimum time or space an algorithm requires as input size approaches infinity. When we say an algorithm is Ω(f(n)), we mean that its running time grows at least as fast as f(n) for sufficiently large n. For example, Ω(n) means the algorithm must perform at least a linear amount of work regardless of the input. Omega notation is useful for establishing that no algorithm can solve a particular problem faster than a certain rate, providing a theoretical limit on optimization.

Theta Notation (Θ) - Tight Bound

Theta notation describes both the upper and lower bounds simultaneously, providing a tight bound on an algorithm's growth rate. When we say an algorithm is Θ(f(n)), we mean that its running time grows at exactly the rate of f(n) for sufficiently large n. it is both O(f(n)) and Ω(f(n)). This indicates that f(n) accurately characterizes the algorithm's asymptotic behavior. For example, if an algorithm is Θ(n log n), its running time will consistently grow proportionally to n log n in all cases (best, average, and worst). Theta notation provides the most precise characterization of an algorithm's performance when the upper and lower bounds match.

(d)

Big-O notation is most commonly used because it focuses on **worst case performance**, which is critical for practical applications where systems must be reliable and predictable. Developers and engineers need to guarantee that an algorithm will complete within acceptable time limits even under the most unfavorable conditions. Designing systems based on worst-case bounds ensures they won't fail or become unacceptably slow when encountering difficult inputs, making Big-O notation essential for setting performance guarantees and making safe design decisions in real-world software development.

(e)

Internal Sorting vs External Sorting

Internal Sorting

Internal sorting is performed entirely in the computer's main memory (RAM). All data to be sorted fits into the available RAM, allowing the algorithm to access any element directly and quickly. This method is fast because it avoids slow disk I/O operations and takes advantage of rapid random access to memory locations.

Example: **QuickSort** - A divide-and-conquer algorithm that recursively partitions an array around pivot elements. It operates entirely in RAM, swapping elements in-place and making multiple passes through the data. QuickSort is ideal when the entire dataset (such as sorting a few thousand records) can comfortably fit in memory.

External Sorting

External sorting is used when the dataset is too large to fit entirely in main memory, requiring data to be stored on external storage devices like hard drives or SSDs. The algorithm must read data from disk in chunks, sort these chunks in memory, write them back to disk, and then merge them. This approach is necessary for handling massive datasets but is slower due to the significant overhead of disk I/O operations.

Example: **External Merge Sort** - This algorithm divides the large dataset into smaller chunks that fit in memory, sorts each chunk individually using an internal sorting algorithm, writes the sorted chunks to disk, and then performs a multi-way merge of these sorted chunks to produce the final sorted output. It's commonly used for sorting database files containing millions or billions of records that exceed available RAM.

(f)

Adaptive Sorting Algorithm

An adaptive sorting algorithm is one that takes advantage of existing order in the input data to improve its performance. It runs faster when the input is partially or fully sorted compared to when the input is in random or reverse order. The algorithm adapts its behavior based on the degree of preexisting order, becoming more efficient as the input becomes more organized.

Adaptive Sorting Algorithm Example:

Insertion Sort - This algorithm is highly adaptive because it performs fewer comparisons and shifts when encountering already-sorted or nearly-sorted data. In the best case (fully sorted input), it runs in O(n) time, simply confirming that each element is already in the correct position. However, with completely unsorted data, it degrades to $O(n^2)$ performance.

Non-Adaptive Sorting Algorithm Example:

Selection Sort - This algorithm always performs the same number of comparisons regardless of the input's initial order. Even if the array is already sorted, Selection Sort still scans through the remaining unsorted portion to find the minimum element for each position, resulting in $O(n^2)$ time complexity in all cases (best, average, and worst). It doesn't benefit from any pre-existing order in the data.

(g)

Bubble Sort with Ascending Order

Initial Array: [25, 12, 9, 30, 18]

Pass 1:

- `Compare 25 and 12 → Swap → [12, 25, 9, 30, 18]`
- `Compare 25 and 9 → Swap → [12, 9, 25, 30, 18]`
- `Compare 25 and 30 → No swap → [12, 9, 25, 30, 18]`
- `Compare 30 and 18 → Swap → [12, 9, 25, 18, 30]`

Result after Pass 1: [12, 9, 25, 18, 30]

Pass 2:

- `Compare 12 and 9 → Swap → [9, 12, 25, 18, 30]`
- `Compare 12 and 25 → No swap → [9, 12, 25, 18, 30]`
- `Compare 25 and 18 → Swap → [9, 12, 18, 25, 30]`
- `Compare 25 and 30 → No swap → [9, 12, 18, 25, 30]`

Result after Pass 2: [9, 12, 18, 25, 30]

Pass 3:

- `Compare 9 and 12 → No swap → [9, 12, 18, 25, 30]`
- `Compare 12 and 18 → No swap → [9, 12, 18, 25, 30]`
- `Compare 18 and 25 → No swap → [9, 12, 18, 25, 30]`
- `Compare 25 and 30 → No swap → [9, 12, 18, 25, 30]`

Result after Pass 3: [9, 12, 18, 25, 30]

Pass 4:

- `Compare 9 and 12 → No swap → [9, 12, 18, 25, 30]`
- `Compare 12 and 18 → No swap → [9, 12, 18, 25, 30]`
- `Compare 18 and 25 → No swap → [9, 12, 18, 25, 30]`

Result after Pass 4: `[9, 12, 18, 25, 30]`

Final Sorted Array: `[9, 12, 18, 25, 30]`

The array is now sorted in ascending order. Note that after Pass 2, the array was already sorted, but Bubble Sort continued to verify this through subsequent passes (or would stop early with an optimized version that detects no swaps occurred).

(h)

Time Complexities of Bubble Sort and Selection Sort

Bubble Sort

- **Worst-case time complexity:** $O(n^2)$
  - Occurs when the array is sorted in reverse order, requiring maximum comparisons and swaps
- **Best-case time complexity:** $O(n)$
  - Occurs when the array is already sorted; with optimization (early termination when no swaps occur), it makes only one pass through the data
- **Average-case time complexity:** $O(n^2)$
  - On average, with randomly ordered data, Bubble Sort requires approximately $n^2/2$ comparisons

Selection Sort

- **Worst-case time complexity:** $O(n^2)$
  - Selection Sort always scans the entire unsorted portion to find the minimum element
- **Best-case time complexity:** $O(n^2)$
  - Even with a sorted array, Selection Sort performs the same number of comparisons to find the minimum in each pass
- **Average-case time complexity:** $O(n^2)$
  - The number of comparisons remains constant regardless of input arrangement

Key Difference: Selection Sort is non-adaptive and maintains $O(n^2)$ complexity in all cases, while Bubble Sort can achieve $O(n)$ in the best case when optimized to detect an already-sorted array.

(i)

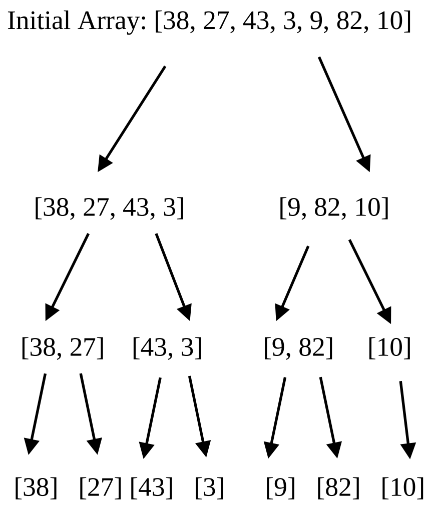
Merge Sort and the Divide and Conquer Strategy

Merge Sort is a classic example of the divide-and-conquer paradigm, which breaks down a problem into smaller sub problems, solves them independently, and combines their solutions. Let me illustrate this with a concrete example.

Example Array: [38, 27, 43, 3, 9, 82, 10]

1. Divide Phase

The algorithm recursively divides the array into two halves until each subarray contains only a single element:



At this point, we have seven subarrays of size 1, which are all considered sorted.

2. Conquer Phase

The algorithm begins solving the subproblems by recognizing that single-element arrays are already sorted. Now it's ready to start merging.

3. Combine (Merge) Phase

The algorithm merges pairs of sorted subarrays back together in sorted order:

Level 1 Merging (merge pairs of single elements):

- `Merge [38] and [27] → Compare 38 and 27 → [27, 38]`
- `Merge [43] and [3] → Compare 43 and 3 → [3, 43]`
- `Merge [9] and [82] → Compare 9 and 82 → [9, 82]`
- `[10] remains as is`

Result: [27, 38], [3, 43], [9, 82], [10]

Level 2 Merging (merge pairs of two-element arrays):

- Merge [27, 38] and [3, 43]:
    - `Compare 27 and 3 → take 3 → [3]`
    - `Compare 27 and 43 → take 27 → [3, 27]`
    - `Compare 38 and 43 → take 38 → [3, 27, 38]`
    - `Take remaining 43 → [3, 27, 38, 43]`
- Merge [9, 82] and [10]:
    - `Compare 9 and 10 → take 9 → [9]`
    - `Compare 82 and 10 → take 10 → [9, 10]`
    - `Take remaining 82 → [9, 10, 82]`

Result: [3, 27, 38, 43], [9, 10, 82]

Level 3 Merging (final merge):

- Merge [3, 27, 38, 43] and [9, 10, 82]:
    - `Compare 3 and 9 → take 3 → [3]`
    - `Compare 27 and 9 → take 9 → [3, 9]`
    - `Compare 27 and 10 → take 10 → [3, 9, 10]`
    - `Compare 27 and 82 → take 27 → [3, 9, 10, 27]`
    - `Compare 38 and 82 → take 38 → [3, 9, 10, 27, 38]`
    - `Compare 43 and 82 → take 43 → [3, 9, 10, 27, 38, 43]`
    - `Take remaining 82 → [3, 9, 10, 27, 38, 43, 82]`

Final Sorted Array: [3, 9, 10, 27, 38, 43, 82]

This divide-and-conquer approach gives Merge Sort its O(n log n) time complexity: the array is divided $\log_2(7) \approx 3$ times (divide phase), and at each level of recursion, merging all elements takes O(n) time (combine phase), resulting in O(n log n) overall performance.

(j)

Quick Sort Using Last Element as Pivot

Initial Array: [14, 7, 21, 10, 5]

Partition 1:

Pivot = 5 (last element)

- Start with i = -1 (index before the first element)
- j scans from index 0 to 3

Scanning process:

- j = 0: Compare 14 with 5 → 14 > 5, no swap, i stays at -1
- j = 1: Compare 7 with 5 → 7 > 5, no swap, i stays at -1
- j = 2: Compare 21 with 5 → 21 > 5, no swap, i stays at -1
- j = 3: Compare 10 with 5 → 10 > 5, no swap, i stays at -1

After scanning, i = -1, so increment i to 0 and swap element at i (14) with pivot (5):

- Swap arr[0] and arr[4] → [5, 7, 21, 10, 14]

Pivot 5 is now at index 0 (final position)

Array after Partition 1: [5, 7, 21, 10, 14]

Recursively sort:

- Left subarray: [] (empty, no elements before index 0)
- Right subarray: [7, 21, 10, 14] (elements after index 0)

Partition 2: (Right subarray [7, 21, 10, 14])

Pivot = 14 (last element)

- `Start with i = 0 (relative to subarray, actual index 1)`
- `j scans from index 1 to 3 (relative to subarray)`

Scanning process:

- `j = 1: Compare 7 with 14 → 7 < 14, increment i to 1, swap arr[1] with arr[1] → [5, 7, 21, 10, 14]`
- `j = 2: Compare 21 with 14 → 21 > 14, no swap, i stays at 1`
- `j = 3: Compare 10 with 14 → 10 < 14, increment i to 2, swap arr[2] with arr[3] → [5, 7, 10, 21, 14]`

After scanning, i = 2, increment i to 3 and swap element at i (21) with pivot (14):

- `Swap arr[3] and arr[4] → [5, 7, 10, 14, 21]`

Pivot 14 is now at index 3 (final position)

Array after Partition 2: [5, 7, 10, 14, 21]

Recursively sort:

- `Left subarray: [7, 10] (elements before index 3)`
- `Right subarray: [21] (single element, already sorted)`

Partition 3: (Left subarray [7, 10])

Pivot = 10 (last element)

- `Start with i = 0 (relative to subarray, actual index 1)`
- `j scans from index 1 (relative to subarray)`

Scanning process:

- `j = 1: Compare 7 with 10 → 7 < 10, increment i to 1, swap arr[1] with arr[1] → [5, 7, 10, 14, 21]`

After scanning, i = 1, increment i to 2 and swap element at i (10) with pivot (10):

- `Swap arr[2] and arr[2] → [5, 7, 10, 14, 21] (no change)`

Pivot 10 is now at index 2 (final position)

Array after Partition 3: [5, 7, 10, 14, 21]

Recursively sort:

- `Left subarray: [7] (single element, already sorted)`
- `Right subarray: [] (empty)`

Final Sorted Array: [5, 7, 10, 14, 21]

Summary of Partitions:

- `Partition 1: [5, 7, 21, 10, 14] - pivot 5 placed`
- `Partition 2: [5, 7, 10, 14, 21] - pivot 14 placed`
- `Partition 3: [5, 7, 10, 14, 21] - pivot 10 placed`