Algorithms - Chapters 1–3 Expanded Study Notes

# Chapter 1: Introduction to Algorithms

- \*\*Definition\*\*: An algorithm is a precise, step-by-step computational procedure that takes an input, processes it, and produces an output.  
- \*\*Key Properties\*\*:  
 1. \*\*Finite\*\* – Must end after a certain number of steps.  
 2. \*\*Unambiguous\*\* – Every step is clearly defined.  
 3. \*\*Effective\*\* – Each step is basic enough to be carried out.  
 4. \*\*Efficient\*\* – Optimizes resources like time and memory.  
- \*\*History & Contributors\*\*:  
 - Al-Khwarizmi: Early algorithmic methods in arithmetic and algebra.  
 - Alan Turing: Defined computational limits via the Turing Machine.  
 - Donald Knuth: Formalized algorithms in "The Art of Computer Programming."  
- \*\*Applications\*\*:  
 - Search Engines: Indexing, ranking.  
 - GPS: Shortest path algorithms.  
 - Logistics: Optimal routing and scheduling.  
 - Stable Marriage Problem: Matching pairs optimally.

Example: Iterative multiplication in Python:

def multiply(m, n):  
 result = 0  
 for \_ in range(n):  
 result += m  
 return result  
  
print(multiply(121, 234)) # Outputs the product of 121 and 234

# Chapter 2: Recursion and Problem Solving

- \*\*Recursion\*\*: Solving a problem by reducing it to smaller instances of the same problem.  
- \*\*Structure\*\*:  
 1. \*\*Base Case\*\* – Stops recursion.  
 2. \*\*Recursive Case\*\* – Calls the function on a smaller input.  
- \*\*Advantages\*\*:  
 - Elegant and often closely matches mathematical formulas.  
 - Useful for divide-and-conquer strategies.  
- \*\*Disadvantages\*\*:  
 - Can be inefficient (stack overhead).  
 - May cause stack overflow if not careful.  
- \*\*Examples\*\*:  
 - Factorial computation.  
 - Fibonacci sequence.  
 - Tree traversals.

Example: Recursive factorial in Python:

def factorial(n):  
 if n == 0:  
 return 1  
 return n \* factorial(n-1)  
  
print(factorial(5)) # Output: 120

# Chapter 3: Sorting Algorithms

- \*\*Purpose\*\*: Arrange elements into a specific order (ascending/descending).  
- \*\*Common Sorting Algorithms\*\*:  
 1. \*\*Insertion Sort\*\* – Builds sorted array one element at a time; O(n²) worst-case.  
 2. \*\*Merge Sort\*\* – Divide-and-conquer, splits array, sorts halves, and merges; O(n log n) worst-case.  
 3. \*\*Quicksort\*\* – Picks a pivot, partitions, sorts recursively; O(n log n) average, O(n²) worst-case if pivots are poor.  
- \*\*Stability\*\*: Whether equal elements keep their relative order after sorting.  
- \*\*In-place vs. Not\*\*:  
 - Merge Sort: Requires extra memory.  
 - Quicksort: Can be in-place.

Example: Merge Sort in Python:

def merge\_sort(arr):  
 if len(arr) > 1:  
 mid = len(arr) // 2  
 L, R = arr[:mid], 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  
  
data = [38, 27, 43, 3, 9, 82, 10]  
merge\_sort(data)  
print(data) # Output: [3, 9, 10, 27, 38, 43, 82]