UNIT I ROLE OF ALGORITHMS IN COMPUTING & COMPLEXITY ANALYSIS

Algorithms – Algorithms as a Technology – Time and Space complexity of algorithms – Asymptotic analysis – Average and worst-case analysis – Asymptotic notation – Importance of efficient algorithms – Program performance measurement – Recurrences: The Substitution Method – The Recursion – Tree Method – Data structures and algorithms.

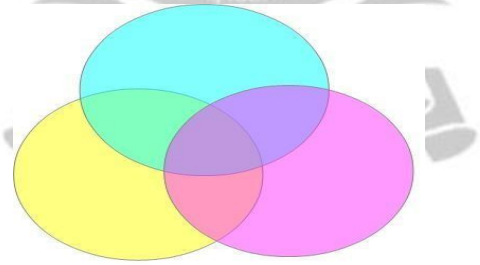
IMPORTANCE OF EFFICIENT ALGORITHMS

In computer science, algorithmic efficiency is a characteristic of an algorithm that is related to the number of computational resources required by the algorithm. An algorithm's resource use must be evaluated, and the efficiency of an algorithm may be assessed based on the use of various resources. For a recurring or continuous process, algorithmic efficiency is comparable to engineering performance. We want to use as few resources as possible to maximize efficiency. However, because various resources, such as time and space complexity, cannot be directly compared, which of the proposed algorithms is judged to be more efficient typically relies on whatever efficiency metric is considered more significant.

Efficiency Factors

Efficiency is dependent on two factors:

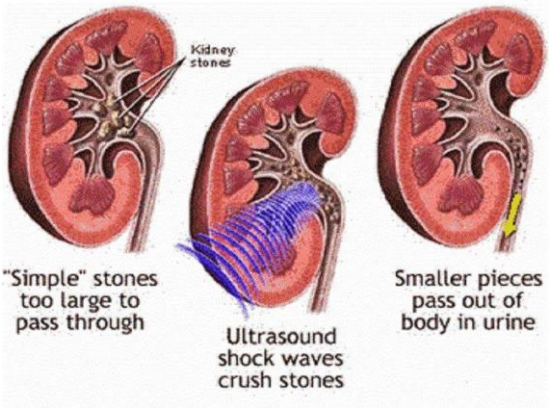
Space Efficiency



In some cases, the amount of space/memory consumed has to be examined. For example, in dealing with huge amounts of data or in programming embedded systems, memory consumed must be analyzed.

Space/memory usage components are:

- **Space of Instruction:** Compiler, compiler settings, and target machine (CPU), all have an impact on space of instruction.



- **Data Space:** Data size/dynamically allocated memory, static program variables are all factors affecting data space.
- **Stack Space at Runtime:** Compiler, run-time function calls and recursion, local variables, and arguments all have an impact on stack space at runtime.

Time Efficiency

Obviously, the faster a program/function completes its objective, the better the algorithm. The actual running time is determined by a number of factors:

- **Computer's Speed:** processor (not simply clock speed), I/O, and so on.
- **Compiler:** The compiler, as well as the compiler parameters.
- **Amount of Data:** The amount of data, for example, whether to search a lengthy or a short list.
- **Actual Data:** The actual data, such as whether the name is first or last in a sequential search.

Approaches to Time Efficiency

There are two ways to assess time complexity:

Asymptotic Categorization:

This employs basic categories to offer a basic notion of performance, which is also called an order of magnitude. If the algorithms are similar, the data amount is little, or performance is essential, then the next method can be explored further. Estimation of running time depends on two things, i.e., Code Analysis and Code Execution, which are elaborated below:

1. **Code Analysis:** We can accomplish the following things by analyzing the code:

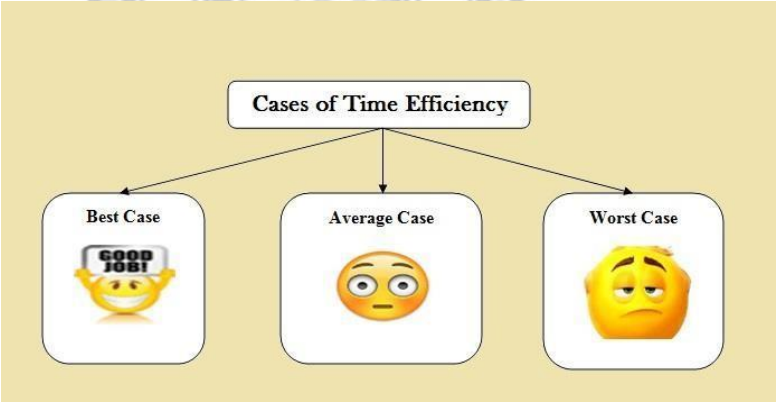
- **Operation Counts:** Choose the most frequently performed operation(s) and count how many times each one is performed.
- **Step Counts:** Calculate the total number of steps and potentially lines of code that program executes.

2. **Code Execution:** We may accomplish the following by executing the code:

```
27  1                def merge_sort(m):
28                       """
29                       Return a sorted copy of m
30                       Uses the recursive merge sort algorithm
31                       """
32
33  75001                if len(m) <= 1:
34  37500                    return m
35  37501                middle = len(m) // 2
36  37501                left = m[:middle]
37  37501                right = m[middle:]
38  37501                return merge(
39  37501                    merge_sort(left), merge_sort(right)
40                       )
41
42  1                def merge(left, right):
43  37498                    result = []
44  37498                    left_idx, right_idx = 0, 0
45  518646                   while left_idx < len(left) and right_idx < len(right):
46  481149                       if left[left_idx] <= right[right_idx]:
47  237261                           result.append(left[left_idx])
48  237261                           left_idx += 1
49                           else:
50  243887                       result.append(right[right_idx])
51  243887                       right_idx += 1
52  37497                    if left_idx < len(left):
53  16843                        result.extend(left[left_idx:])
54  37497                    if right_idx < len(right):
55  20654                        result.extend(right[right_idx:])
56  37497                return result
```

- **Benchmarking:** Running the software on a variety of data sets and comparing the results.
- **Description:** A report on the number of hours spent in each routine of a program, which is used to identify and eliminate the program's hot spots. This perception is frequently questioned. Although other description modes give data in units other than time (for example, call counts) and/or at levels of granularity besides per-routine, the concept is the same.

Three Cases of Time Efficiency



- Worst case
- Average case
- Best case

The worst-case scenario is evaluated since it provides an upper limit on projected performance. Furthermore, the average scenario is typically more difficult to establish (it is more data dependent), and it is frequently the same as the worst case. It may be essential to examine all three situations on some occasions.