**Fundamentals of the Analysis of Algorithm Efficiency**

**The Analysis Framework**

* Time efficiency, also called time complexity, indicates how fast an algorithm in question runs.
* Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

**Measuring an Input’s Size**

* Almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on.
* Algorithm’s efficiency is measured as a function of some parameter ‘n’ indicating the algorithm’s input size.
* For example, it will be the size of the list for problems of sorting, searching, finding the list’s smallest
* element, and most other problems dealing with lists.
* For the problem of evaluating a polynomial p(x) = anxn + . . . + a0 of degree n, it will be the polynomial’s degree or the number of its coefficients, which is larger by 1 than its degree.
* Computing the product of two n × n matrices. There are two natural measures of size for this problem. One is the matrix order n, and the other one is the total number of elements N in the matrices being multiplied.
* For a spell-checking algorithm, if the algorithm examines individual characters of its input, efficiency is measured by the number of characters; if it works by processing words, efficiency is measured by counting the number of words in the input.
* Problems such as checking primality of a positive integer n, efficiency is measured by the number b of bits in the n’s binary representation:



**Units for Measuring Running Time**

* Identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
* Sorting algorithms - the basic operation is a key comparison.
* Algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction.
* Let *cop* be the execution time of an algorithm’s basic operation on a particular computer, and let *C(n)* be the number of times this operation needs to be executed. Running time *T (n)* of a program is calculated by *T (n)* ≈ *copC(n).*
* *C(n)* = 1/2 *n(n* − 1*)*, how much longer will the algorithm run if the input size is doubled? The answer is about four times longer.

****

**Orders of Growth**

****

* The function log2 n increases in value by just 1 (because log2 2n = log2 2 + log2 n = 1+ log2 n);
* The linear function increases twofold.
* The linearithmic function n log2 n increases slightly more than twofold.
* The quadratic function n2 and cubic function n3 increase fourfold and eightfold, respectively (because (2n)2 = 4n2 and (2n)3 = 8n3);
* The value of 2n gets squared (because 22n = (2n)2), and n! increases much more than that.

**Worst-Case, Best-Case, and Average-Case Efficiencies**

ALGORITHM SequentialSearch(A[0..n − 1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n − 1] and a search key K

//Output: The index of the first element in A that matches K

// or −1 if there are no matching elements

i ←0

while i < n and A[i] != K do

i ←i + 1

if i < n return i

else return −1

**Basic Operation:** Comparison

**Worst-case efficiency**

* The input for which the algorithm runs the longest among all possible inputs of that size.

Cworst(n) = n.

**Best-case efficiency**

* The input for which the algorithm runs the fastest among all possible inputs of that size.

Cbest(n) = 1

**Average-case efficiency**

****

* For example, if p = 1 (the search must be successful), the average number of key comparisons made by sequential search is (n + 1)/2; that is, the algorithm will inspect, on average, about half of the list’s elements.
* If p = 0 (the search must be unsuccessful), the average number of key comparisons will be ‘n’ because the algorithm will inspect all ‘n’ elements on all such inputs.

**Amortized efficiency**

* It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.
* In some situations a single operation can be expensive, but the total time for an entire sequence of ‘n’ such operations is always significantly better than the worst-case efficiency of that single operation multiplied by n.