8. MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS

1.1 General Plan for Analyzing the Time Efficiency of Non recursive Algorithms:

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation* (in the inner most oop).
3. Check whether the ***number of times the basic operation is executed*** depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case *efficiencies* have to be investigated separately.
4. Set up a *sum* expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its *order of growth*.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

```
ALGORITHM Max Element(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    Max val ←A[0]
    for i ←1 to n − 1 do
        if A[i]>maxval
            maxval←A[i]
    return maxval
```

Algorithm analysis

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body:
    - The comparison A[i]> maxval and
    - The assignment max val←A[i].

- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n;

therefore, there is no need to distinguish among the worst, average, and best cases here.

- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

$$() = \sum_{=}^{-}$$

i.e., Sum up 1 in repeated n-1 times

$$() = \sum_{=}^{-} = - \in ()$$

EXAMPLE 2: Consider the **element uniqueness problem**: check whether all the Elements in a given array of n elements are distinct.

ALGORITHM Unique Elements (A[0..n − 1])

```
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct and "false"
    otherwise
    for i ←0 to n − 2 do
        for j ←i + 1 to n − 1 do
            if A[i]= A [j ] return false
return true
```

Algorithm Analysis

- The natural measure of the input's size here is again n (the number of elements in the array).
- Sincetheinnermostloopcontainsasingleoperation(thecomparisonoftwoelements), we should consider it as the algorithm's basic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n −2.

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i) \\
&= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).
\end{aligned}
$$

EXAMPLE 3: Consider matrix multiplication. Given two n × n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C = AB. By definition, C

an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:
where C[i, j]= A[i, 0]B[0, j]+ . . . + A[i, k]B[k, j]+ . . . + A[i, n − 1]B[n − 1, j] for every pair of indices 0 ≤ i, j ≤ n − 1.

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two square matrices of order n by the definition-based
    algorithm
    //Input: Two n × n matrices A and B
    //Output: Matrix C = AB
    for i ←0 to n − 1 do
        for j ←0 to n − 1 do
            C[i, j ]←0.0
            for k←0 to n − 1 do
                C[i, j ]←C[i, j ]+ A[i, k] ∗ B[k, j]
    return C
```

Algorithm analysis

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n −1.

- Therefore, the number of multiplications made for every pair of specific

$$\sum_{k=0}^{n-1} 1.$$

values of variables i and j is

The total number of multiplications M(n) is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1$$

Now, we can compute this sum by using formula (S1) and rule (R1)

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

.

The running time of the algorithm on a particular machine m, we can do

$$T(n) \approx c_m M(n) = c_m n^3,$$

it by the product If we consider, time spent on the additions too, then

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3$$

the total time on the machine is

Example: 4

The following algorithm finds the number of binary digits in the **binary representation** of a positive decimal integer.

ALGORITHM Binary(n)

```
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary
    representation count ←1
    while n > 1 do
        count
        ←count +
        1 n← ⌊n/2]
return count
```

Algorithm Analysis:

- An input’s size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
- The exact formula for the number of times.
- The comparison $n > 1$ will be executed is actually $\lfloor \log_2 n] +1$.