Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

What is an algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem,
i.e., for obtaining a required output for any legitimate input in a finite amount of time
Important Points about Algorithms

Characteristics of Algorithm

- Input: Zero or more quantities are externally supplied
- Definiteness: Each instruction is clear and unambiguous
- Finiteness: The algorithm terminates in a finite number of steps.
- Effectiveness: Each instruction must be primitive and feasible
- Output: At least one quantity is produced

Why do we need Algorithms?

- It is a tool for solving well-specified Computational Problem.
- Problem statement specifies in general terms relation between input and output
- Algorithm describes computational procedure for achieving input/output relationship This Procedure is irrespective of implementation details

Why do we need to study algorithms?

Exposure to different algorithms for solving various problems helps develop skills to design algorithms for the problems for which there are no published algorithms to solve it

Two descriptions of Euclid’s algorithm

Natural Language

Euclid’s algorithm for computing gcd(m,n)

Step 1 If n = 0, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

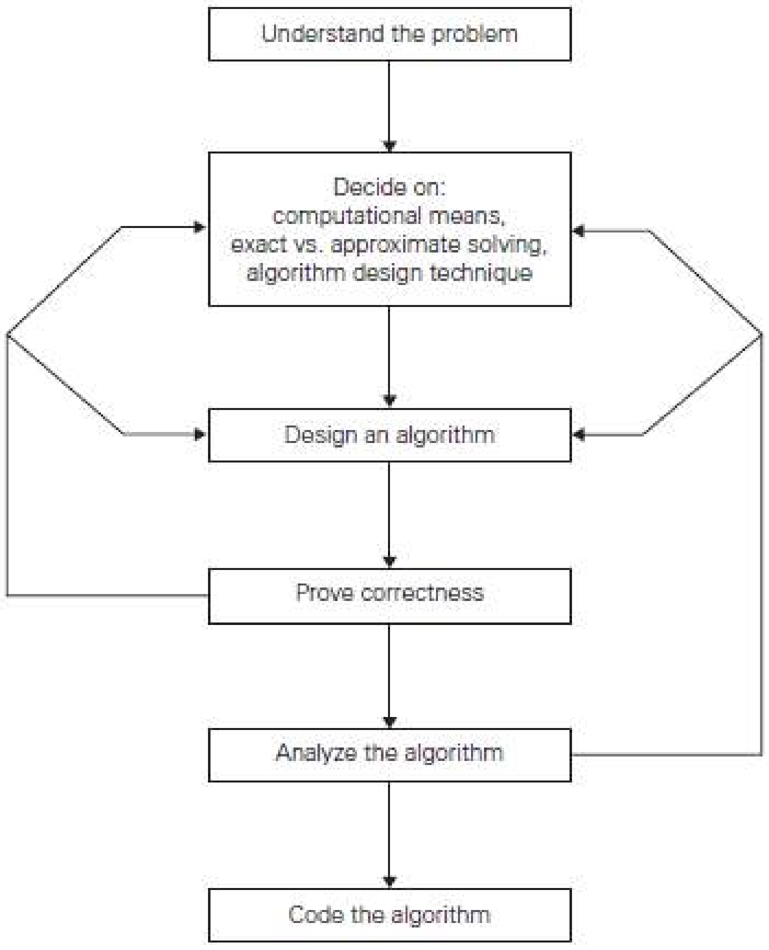
Step 3 Assign the value of n to m and the value of r to n. Go to step 1.

Pseudo Code

```
ALGORITHM Euclid(m,n)
//computes gcd(m,n) by Euclid’s method
//Input: Two nonnegative,not both zero integers
//Output:Greatest common divisor of m and n
while n ≠ 0 do
    r ← m mod n
   m← n
   n ← r
return m
```

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Fundamentals of Algorithmic Problem Solving



Understand the Problem

- Algorithms are procedural solutions to problems.
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- Boundary conditions should be clearly understood

Decide on computational means

- Sequential vs Parallel algorithm
- Exact vs Approximation algorithm
- Data Structures + Algorithms = Programs

Design Algorithm

Specifying algorithm

- Natural Language
- Pseudo code
- Flowchart

Correctness:

Mathematical Induction

Exact Algorithms: correct algorithm is the one that works for all legitimate inputs.

Approximate Algorithms: Error in tolerance limit

Analyzing Algorithm

- Time vs Space efficiency
- Simplicity vs Generality

Coding the algorithm

- Testing
- Debugging
- Code optimization

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Important Problem Types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Sorting

The sorting problem is to rearrange the elements of a given list in non-decreasing (ascending) or decreasing order (descending) order.

- Examples of sorting algorithms
    - Selection sort
    - Bubble sort
    - Insertion sort
    - Merge sort
    - Heap sort …

Number of key comparisons is used to determine time complexity of sorting algorithms

Two properties related to sorting algorithms

- Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.

- In place: A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Searching

Find a given value, called a search key, in a given set.

Examples of searching algorithms

- Sequential searching
- Binary searching…

String Matching

A string is a sequence of characters from an alphabet.

Text strings: letters, numbers, and special characters.

String matching: searching for a given word/pattern in a text.

Text: I am a computer science graduate

Pattern: computer

Graph problems

A graph is a collection of points called vertices and edges

Examples of graph problems are graph traversal, traveling salesman problem, shortest path algorithm, topological sort,and the graph-coloring problem

Combinatorial problems

These are problems for which it is required to generate permutations, a combinations, or a subset that satisfies certain constraints.

A desired combinatorial object may have an associated cost that needs to be minimized or maximized

In practical, the combinatorial problems are the most difficult problems in computing.

The traveling salesman problem and the graph coloring problem are examples of combinatorial problems.

Geometric problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons.

Geometric algorithms are used in computer graphics, robotics etc.

Examples: closest-pair problem and the convex-hull problem

Numerical problems

Numerical problems are problems that involve computing definite integrals, evaluating functions, mathematical equations, systems of equations, and so on.

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Textbook Chapter 2
Section 2.1

Analysis of Algorithm

Investigation of Algorithm’s efficiency with respect to two resources time and space is termed as analysis of algorithms.
We need to analyse the algorithms to

- determine the resource requirement(CPU time and memory)
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

There are two approaches to determine time complexity

- Theoretical Analysis
- Experimental study

Theoretical Analysis

General Framework to determine time complexity of algorithm

- Measuring an input's size
- Measuring running time
- Finding Orders of growth
- Worst-base, best-case and average efficiency

Time efficiency is represented as function of input size. Time efficiency is determined by counting the number of times algorithms basic operation executes. This is independent of processor speed, quality of implementation, compiler and etc.

Basic Operation: The operation that contributes most to the running time of an algorithm.
For some problems number of times basic operation executes differs for different inputs of same size for such problems we need to do Best, Worst and Average class analysis

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	Size of list	Key comparison
Multiplication of two matrices	Dimension of matrix	Elementary multiplication

Order of growth of algorithm's running time is important to compare the performance of different algorithms

Best Worst and Average case Analysis

- Worst case Efficiency
    - Number of times basic operation is executed for the worst case input of size n.
    - The algorithm runs the longest among all possible inputs of size n.
- Best case Efficiency
    - Number of times basic operation is executed for the best case input of size n.
    - The algorithm runs the fastest among all possible inputs of size n.
- Average case Efficiency:
    - Number of times basic operation is executed for random input of size n.
    - NOT the average of worst and best case

Time complexity Analysis of Sequential Search

```
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: Returns the index of the first element of 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        //A[I] = K
        return i
    else
        return -1
```

- Worst-Case: Cworst(n) = n
- Best-Case: Cbest(n) = 1

Let **‘p’** be the probability that key is found in the list

Assumption: All positions are equally probable

Case1: key is found in the list

$C_{avg,case1}(n) = p*(1 + 2 + \ldots + n) / n = p*(n + 1) / 2$

Case2: key is not found in the list

$C_{avg, case2}(n) = (1-p)*(n)$

$\mathbf{C_{avg}(n) = p}(n + 1) / 2 \mathbf{ + (1 - p)}(n)$

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Chapter 2 section2.2

Orders of growth of an algorithm’s basic operation count is important

We compare order of growth of functions using asymptotic notations

Asymptotic notations

A way of comparing functions that ignores constant factors and small input sizes

$O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$

$\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$

$\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$

$o(g(n))$: class of functions $f(n)$ that grow *at slower rate* than $g(n)$

$w(g(n))$: class of functions $f(n)$ that grow *at faster rate* than $g(n)$

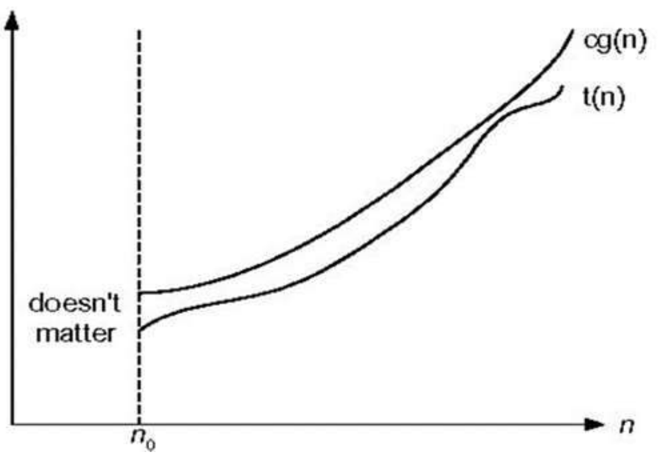
Big O notation

Formal definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$,
if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n,
i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Example: $100n+5 \in O(n)$



Big Omega Notation

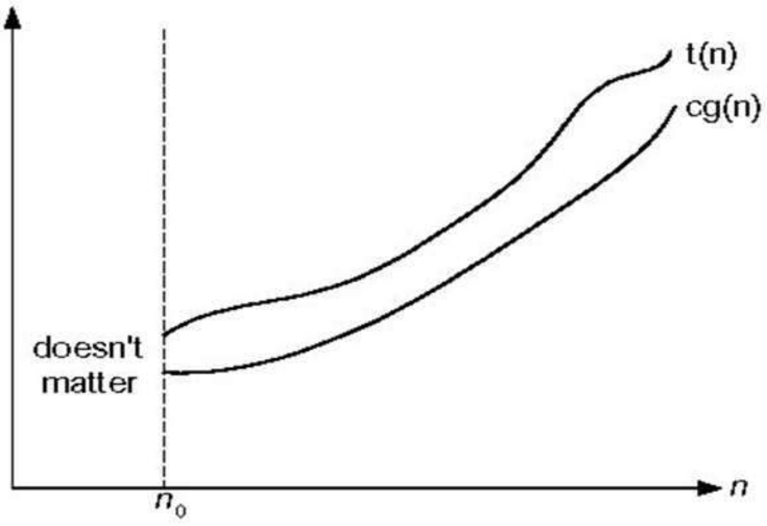
Formal definition

A function *t(n)* is said to be in Ω*(g(n)),* denoted *t(n)* $\in \Omega$*(g(n)),* if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$t(n) \geq cg(n)$ for all $n \geq n_0$

Example: $10n^2 \in \Omega(n^2)$



Theta Notation

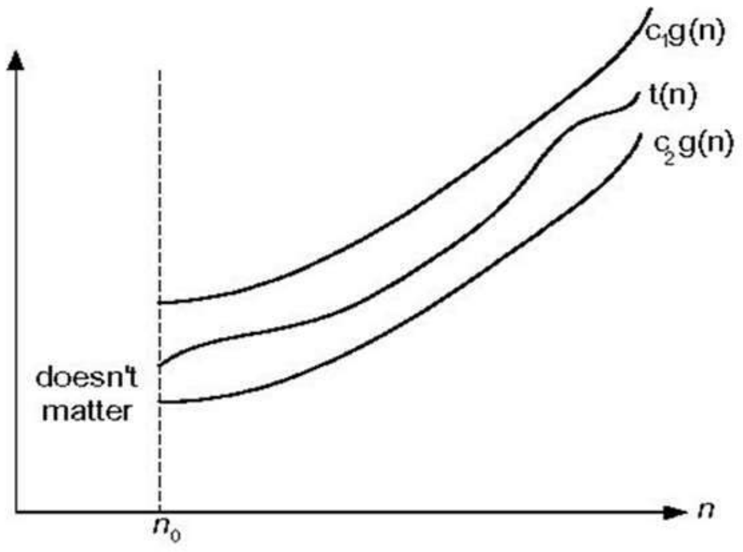
Formal definition

A function *t(n)* is said to be in Θ*(g(n)),* denoted *t(n)* $\in \Theta$*(g(n)),* if *t(n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*,

i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2\, g(n) \le t(n) \le c_1\, g(n) \text{ for all } n \ge n_0$$

Example: $(1/2)n(n-1) \in \Theta(n^2)$



Small o notation

Formal Definition:

A function *t(n)* is said to be in Little-o*(g(n)),* denoted *t(n)* $\in$*o(g(n)),*

if for any positive constant c and some nonnegative integer n_0

$$0 \le t(n) < cg(n) \text{ for all } n \ge n_0$$

Example:

If $f(n) = n$ & $g(n) = n^2$,

then for any value of c>0,

$f(n) < c(n^2)$

$f(n) \in o(g(n))$

Small omega notation

Formal Definition:

A function *t(n)* is said to be in Little- w*(g(n)),* denoted *t(n)* $\in$ w*(g(n)),*

if for any positive constant c and some nonnegative integer n_0

$t(n) > cg(n) \geq 0$ for all $n \geq n_0$

Example : If $f(n) = 3n^2 + 2$, $g(n) = n$

then for any value of c>0

$f(n) > cg(n)$

$f(n) \in w(n)$

Theorems

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

  For example,

  $5n^2 + 3n\log n \in O(n^2)$

- If $t1(n) \in \Theta(g1(n))$ and $t2(n) \in \Theta(g2(n))$, then

  $t1(n) + t2(n) \in \Theta(\max\{g1(n), g2(n)\})$

- $t1(n) \in \Omega(g1(n))$ and $t2(n) \in \Omega(g2(n))$, then

  $t1(n) + t2(n) \in \Omega(\max\{g1(n), g2(n)\})$

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Basic Efficiency Classes to represent time complexity

Class	Name	Example
1	constant	Best case for sequential search
$\log n$	logarithmic	Binary Search
n	linear	Worst case for sequential search
$n \log n$	n-log-n	Mergesort
n^2	quadratic	Bubble Sort
n^3	cubic	Matrix Multiplication
2^n	exponential	Subset generation
$n!$	factorial	TSP using exhaustive search

Using Limits to Compare Order of Growth

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Case1: t(n) $\in$ O(g(n)

Case2: t(n) $\in$ Θ(g(n))

Case3: g(n) $\in$ O(t(n))

t'(n) and g'(n) are first-order derivatives of t(n) and g(n)

L'Hopital's Rule $$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

Stirling's Formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Using Limits to Compare Order of Growth: Example 1

Compare the order of growth of f(n) and g(n) using method of limits

t(n) = 5n3 + 6n + 2 ,   *g(n) = n4*

$$\lim_{n\to\infty}\frac{\mathrm{t}(n)}{g(n)} = \lim_{n\to\infty}\frac{5n^3+6n+2}{n^4} = \lim_{n\to\infty}\left(\frac{5}{n}+\frac{6}{n^3}+\frac{2}{n^4}\right) = 0$$

As per case1

t(n) = O(g(n))

$5n^3 + 6n + 2 = O(n^4)$

Using Limits to Compare Order of Growth: Example 2

$$t(n) = \sqrt{5n^2 + 4n + 2}$$

using the Limits approach determine g(n) such that f(n) = Θ(g(n))

Leading term in square root n2

$$g(n) = \sqrt{n^2} = n$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n\to\infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$

non-zero constant

Hence, t(n) = Θ(g(n)) = Θ(n)

Using Limits to Compare Order of Growth

$$lim_{n \to \infty} t(n)/g(n) \neq 0, \infty \Rightarrow t(n) \in \Theta(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq \infty \quad \Rightarrow t(n) \in O(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq 0 \quad \Rightarrow t(n) \in \Omega(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = 0 \quad \Rightarrow t(n) \in o(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = \infty \quad \Rightarrow t(n) \in \omega(g(n))$$

Using Limits to Compare Order of Growth: Example 3

Compare the order of growth of t(n) and g(n) using method of limits

$t(n) = \log_2 n,\ g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\,\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

$\log_2 n \in o(\sqrt{n})$

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Mathematical Analysis of Non-Recursive algorithms

General Plan for Analysing 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* (The operation that consumes maximum amount of execution time).
3. Check whether the *number of times the basic operation is executed* depends only on the size of an input. If the number of times the basic operation gets executed varies with specific instances (inputs), we need to carry out Best, Worst and Average case analysis
4. Set up a *sum* expressing the number of times the algorithm's basic operation is executed.
5. Simplify the sum using standard formulas and rules , establish its *order of growth*

EXAMPLE 1:

Find the largest element in a list of n numbers.

Assumption: list is implemented as an array.

```
ALGORITHM MaxElement (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
maxval ←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: the number of elements in the array, i.e., n.
- Basic Operation
  There are two operations in the for loop's body:
  - A[i]> maxval
  - Maxval ←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.
- Best /Worst/Average Case
  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 for this problem.
- 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 (inclusively). Hence,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

$\Rightarrow T(n) \in \Theta(n)$

EXAMPLE 2:

Element uniqueness problem:

```
ALGORITHM UniqueElements(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

- Input size: n (the number of elements in the array).
- Basic Operation: Comparison if A[i]= A[j]
- 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. So we need to do best, worst and average case analysis we discuss best and worst case analysis here
  - Best-case situation:

    First two elements of the array are the same

    Number of comparison. Best case = 1 comparison.
  - Worst-case situation:

    The worst- case happens for two-kinds of inputs:

    – Arrays with no equal elements

    – Arrays in which only the last two elements are the pair of equal elements

$$C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2$$

$\Rightarrow \quad T(n)_{worst} \in O(n)$

EXAMPLE 3:

Matrix multiplication.

C=A*B

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 $0 \le i, j \le n - 1$.

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
//Multiplies two square matrices of order n
//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

- Input Size: matrix order n.
- There are two arithmetical operations in the innermost loop, multiplication and addition. But we consider multiplication as the basic operation as multiplication is more expensive as compared to addition
- Total number of elementary multiplications executed by the algorithm 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.

$M(n) \in \Theta(n^3)$

$\Rightarrow$ **$T(n) \in \Theta(n^3)$**

EXAMPLE 4

Determine 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.
- Basic operation Either Division or Addition
- Let us consider addition as basic operation. Number of times addition is executed depends only on the value of n so we don't need to do best, worst and average case analysis separately
- 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, so number of times count ←count + 1 is executed is $\log_2 n + 1$.

  $\Rightarrow T(n) \in (\log_2 n)$

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Mathematical Analysis of Recursive algorithms

General Plan for Analysing 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* (The operation that consumes maximum amount of execution time).
3. Check whether the *number of times the basic operation is executed* depends only on the size of an input. If the number of times the basic operation gets executed varies with specific instances (inputs), we need to carry out Best, Worst and Average case analysis
4. *Set up a recurrence relation*, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence to determine time complexity, establish **order of growth** of its solution

Methods to solve recurrences

- Substitution Method
    - Mathematical Induction
    - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

EXAMPLE 1:

Find n!

$n! = 1 * 2 * \ldots *(n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of n!:

$F(n) = F(n-1) * n$ for $n \geq 1$

$F(0) = 1$

```
ALGORITHM F(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0
    return 1
else
   return F(n − 1) * n
```

Algorithm analysis

Input size: n

Basic operation: Multiplication

Best/Worst/Average case: number of multiplications depend only on n so no best/worst/average case analysis

Recurrence relation and initial condition for number of multiplications required to compute n! is given as

$M(n) = M(n - 1) + 1$ for $n > 0$,

$M(0) = 0$ for $n = 0$.

Method of backward substitutions

$M(n) = M(n - 1) + 1$ substitute $M(n - 1) = M(n - 2) + 1$

$= [M(n - 2) + 1]+ 1$

$= M(n - 2) + 2$ substitute $M(n - 2) = M(n - 3) + 1$

$= [M(n - 3) + 1]+ 2$

$= M(n - 3) + 3$

…

$= M(n - i) + i$

$M(0)=0$ So substitute $i=n$

$= M(n - n) + n$

$= n.$

Therefore $M(n)=n$

$\Rightarrow T(n) \in \Theta(n)$

EXAMPLE 2:

Tower of Hanoi

```
ALGORITHM TOH(n, A, C, B)
//Move disks from source to destination recursively
//Input: n disks and 3 pegs A, B, and C
//Output: Disks moved to destination as in the source order.
    if n=0
        return
    else
        Move top n-1 disks from A to B using C
        TOH(n - 1, A, B, C)
        Move 1 disk from A to C
        Move top n-1 disks from B to C using A
        TOH(n - 1, B, C, A)
```

Algorithm analysis

M(n) = $\mathbf{2M(n-1) + 1}$ for $n > 0$ and $M(0)=0$

$= \mathbf{2^n - 1 \in \Theta(2^n)}$

$\mathbf{\Rightarrow T(n) \in \Theta(2^n)}$

EXAMPLE 3

Determine number of binary digits in the binary representation of a positive decimal integer

```
ALGORITHM BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
    if n = 1
        return 1
    else
        return BinRec(floor(n/2))+ 1
```

Algorithm analysis

Input size=n

Basic operation: Addition

Number of additions depend only on the size of input n so no separate analysis is required for best, worst and average case

Recurrence Relation for number of additions

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

Text Book:
Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin
2nd Edition

Recurrence

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

Recurrences can take many forms

Example:

- T(n)=T(n/2)+1
- T(n)=T(n-1)+1
- T(n)=T(2n/3)+T(n/3)+1

Important Recurrence Types

➢ Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size *n* and a smaller size *n – 1*.

Example: n!

The recurrence equation has the form

$T(n) = T(n-1) + f(n)$

➢ Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size *n* into several smaller instances of size *n/b*, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.

Example: binary search.

The recurrence has the form

$T(n) = aT(n/b) + f(n)$

Methods to solve recurrences

- Substitution Method
    - Mathematical Induction
    - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

Example1:

$T(n) = T(n-1) + 1 \quad n>0 \qquad T(0) = 1$

$T(n) = T(n-1) + 1$

$= T(n-2) + 1 + 1 = T(n-2) + 2$

$= T(n-3) + 1 + 2 = T(n-3) + 3$

…

$= T(n-i) + i$

…

$= T(n-n) + n = n = O(n)$

Example2:

$T(n) \quad = T(n-1) + 2n - 1 \qquad T(0)=0$

$= [T(n-2) + 2(n-1) - 1] + 2n - 1$

$= T(n-2) + 2(n-1) + 2n - 2$

$= [T(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2$

$= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3$

…

$= T(n-i) + 2(n-i+1) + \ldots + 2n - i$

…

= T(n-n) + 2(n-n+1) +...+ 2n - n

= 0 + 2 + 4 +...+ 2n - n

= 2 + 4 +...+ 2n - n

*= 2*n*(n+1)/2 - n*

// arithmetic progression formula 1+...+n = n(n+1)/2 //

= $O(n^2)$

Example 3:

$$T(n) = T(n/2) + 1 \quad n > 1$$
$$T(1) = 1$$

$$T(n) = T(n/2) + 1$$
$$= T(n/2^2) + 1 + 1$$
$$= T(n/2^3) + 1 + 1 + 1$$
$$\ldots\ldots$$
$$= T(n/2^i) + i$$
$$\ldots\ldots$$
$$= T(n/2^k) + k \;\; (k = \log n)$$
$$= 1 + \log n$$
$$= \mathrm{O}(\log \; n)$$

Example 4:

$$T(n) \;\; = 2T(n/2) + cn \qquad n > 1 \quad T(1) = c$$

$$T(n) = 2T(n/2) + cn$$
$$= 2(2T(n/2^2) + c(n/2)) + cn = 2^2T(n/2^2) + cn + cn$$
$$= 2^2(2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3T(n/2^3) + 3cn$$
$$\ldots\ldots$$
$$= 2^iT(n/2^i) + icn$$
$$\ldots\ldots$$
$$= 2^kT(n/2^k) + kcn \quad (k = \log \; n)$$
$$= nT\;(1) + cn\log n = cn + cn\log n$$
$$= \mathrm{O}(n\log \; n)$$

Example 5:

$T(n)=2T(\sqrt{n})+1$ $\quad\quad$ $T(1)=1$

Assume $n=2^m$

Which gives recurrence

$T(2^m)=2T(2^{m/2})+1$

Assume $T(2^m)=S(m)$

Which gives recurrence

$S(m)=2S(m/2)+1$

Solving using backward substitution (reference example3) gives

$S(m)=m+2$

- ⇨ $T(n)=O(logn)$

Performance Analysis Vs Performance Measurement

Performance of an algorithm is measured in terms of time complexity and space complexity

Time complexity Analysis

There are two approaches to determine time complexity

- Theoretical Analysis
- Experimental study/Empirical Analysis

Theoretical Analysis

Theoretical Analysis is evaluation of an Algorithm prior to its implementation on the actual machine so it is machine independent and it helps to compare algorithms irrespective of the machine configuration on which the algorithm is intended to run

Experimental Analysis

This is posterior evaluation of an algorithm. The algorithm is implemented and run on actual machine for different inputs to understand the relation between execution time and input size. This method for determining the performance of an algorithm is machine dependent

Performance Analysis of Sequential Search algorithm.

```
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: Returns the index of the first element of 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 A[i] ≠ K

Basic operation count n (for worst case)

$T(n)_{\text{worst case}} \in O(n)$

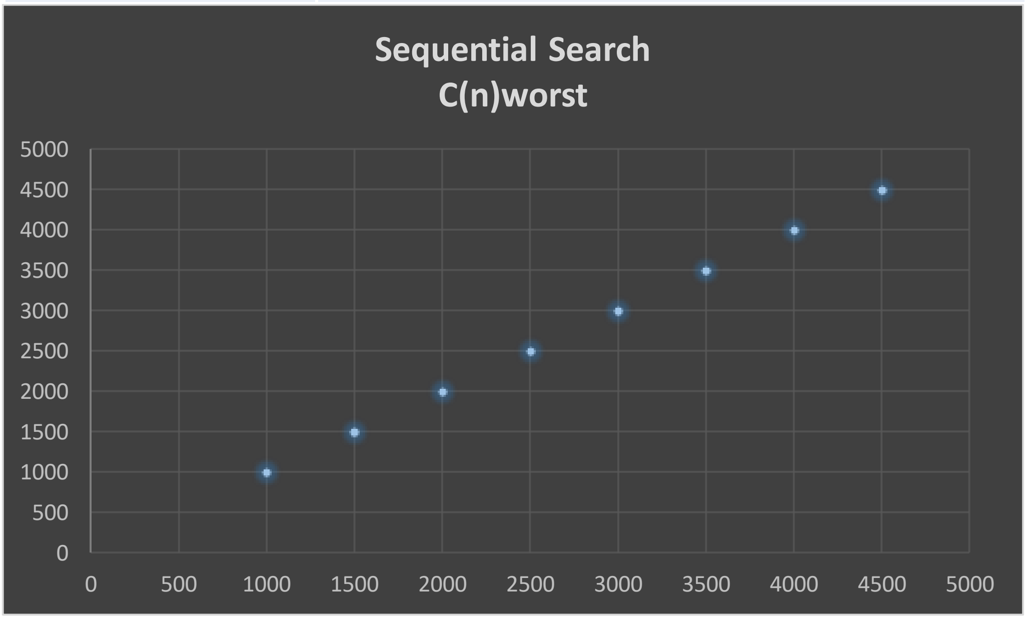
Performance Measurement for sequential search algorithm

```
#include<stdio.h>

#include<stdlib.h>

#include<sys/time.h>

int getrand(int a[], int n)

{

            int i;

             for(i=0;i<n;i++)

             {

                        a[i]=rand()%10000

             }

}

int search(int arr[], int n,int x,int *count)

{

            int i;

            for(i=0;i<n;i++)

            {

        count=count+1;

                      if(arr[i]==x)

                                    return i;

            }

            return -1;

}
```

```
int main()
{
	int a[10000],i,res,count;
	double elapse,start,end;
	struct timeval tv;
	FILE *fp1,*fp2;
	fp1=fopen("seqtime.txt","w");
	fp2=fopen("seqcount.txt","w");
	int key;
	for(i=500;i<=10000;i+=500)// size of the array to be created
	{
		getrand(a,i);
		key=a[i-1];
		count=0;
		gettimeofday(&tv,NULL);
		start=tv.tv_sec+ tv_usec/100000//start time
		res=search(a,i,key,&count);
		gettimeofday(&tv,NULL);
		end=tv.tv_sec+ tv_usec/100000//end time
		elapse=(end-start)*1000
		fprintf(fp1,"%d\t%lf\n",i,elapse);
		fprintf(fp2,"%d\t%d\n",i,count);
	}
	fclose(fp1);
	fclose(fp2);
	return 0;
}
```

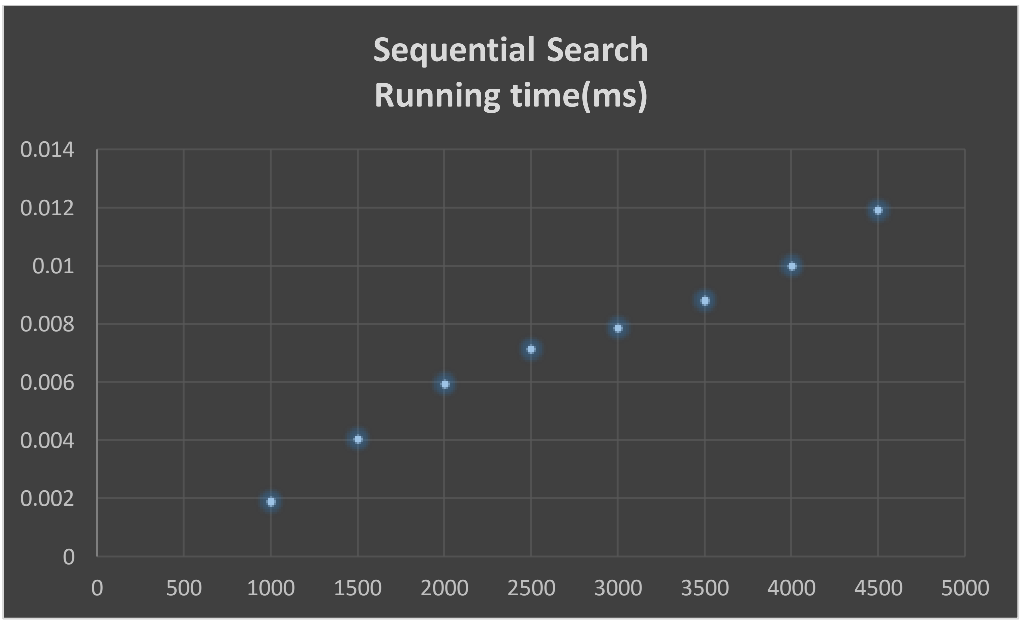
Number of key comparisons for inputs of different sizes

Input Size	Sequential Search C(n)worst
1000	1000
1500	1500
2000	2000
2500	2500
3000	3000
3500	3500
4000	4000
4500	4500



Actual running time for inputs of different sizes

Input Size	Sequential Search Actual Running Time(ms)
1000	0.001907
1500	0.004053
2000	0.00596
2500	0.007153
3000	0.007868
3500	0.008821
4000	0.010014
4500	0.011921



Time complexity (worst case) of sequential search is ***linear*** in terms of length of the list of elements