PES UNIVERSITY

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

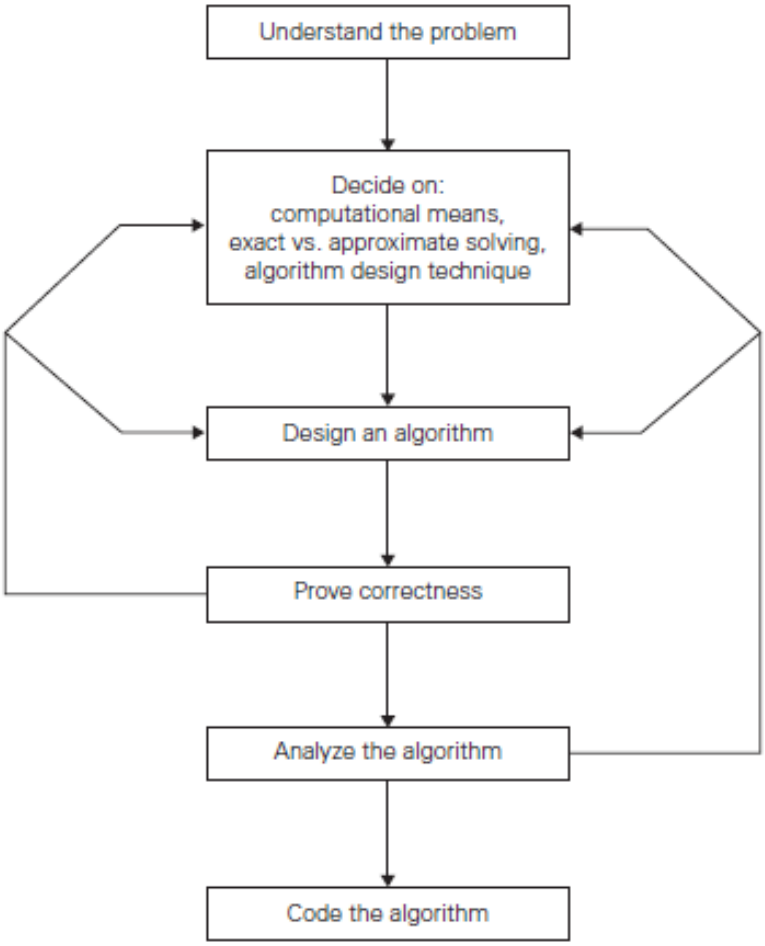
   m← n

   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

PES UNIVERSITY

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.

PES UNIVERSITY

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] ǂ 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

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

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

PES UNIVERSITY

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 $g(n)$

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

Θ ($g(n)$): 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 $g(n)$

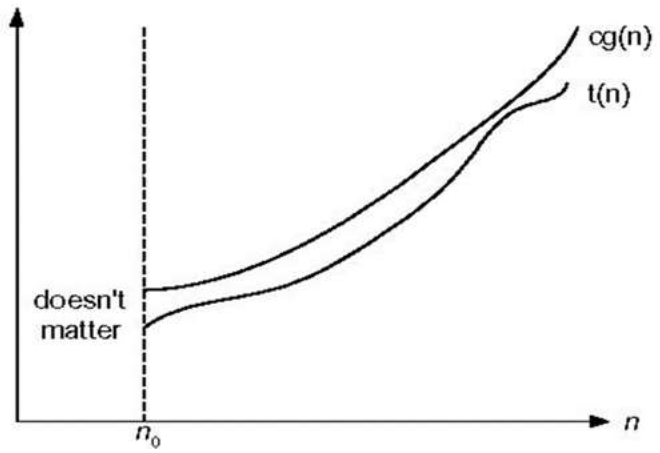
## Big O notation

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

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

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




## Big Omega Notation

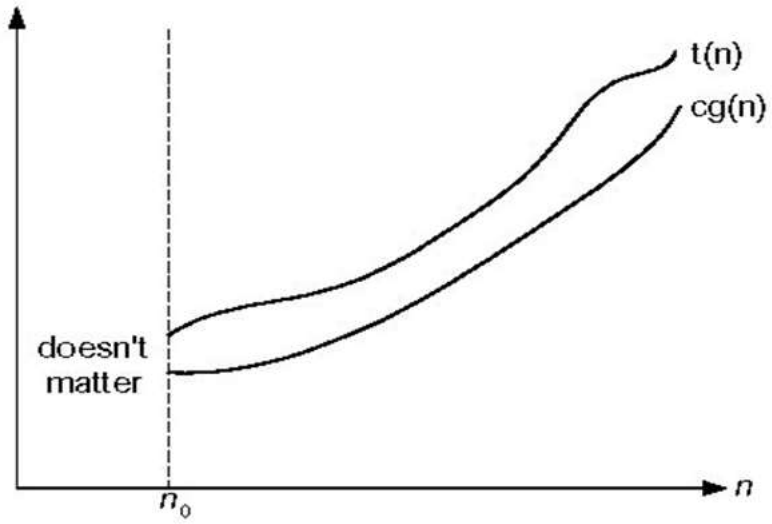
Formal definition

A function *t(n)* is said to be in $\Omega(g(n))$, 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

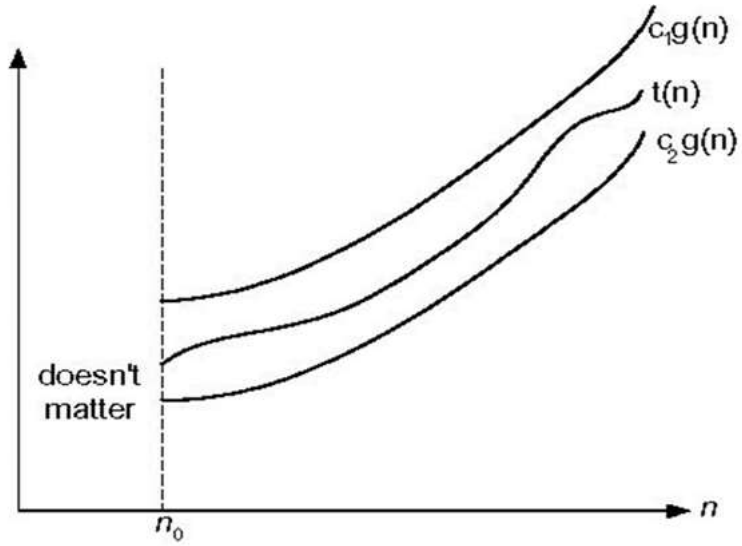
Formal definition

A function *t(n)* is said to be in $\Theta(g(n))$, 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

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

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 $n_0$

$$0 \leq t(n) < cg(n) \text{ for all } n \geq n_0$$

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) ∈ Θ ( g 1 ( n)) and t2 ( n) ∈ Θ ( g2 ( n)) , then

  t1 ( n) + t2 ( n) ∈ Θ(max{ g 1 ( n), g2 ( n)})

- t1(n) ∈ Ω(g1(n)) and t2(n) ∈ Ω(g2(n)), then

  t1(n) + t2(n) ∈ Ω(max{g1(n), g2(n)})

PES UNIVERSITY

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: $t(n) \in \Theta(g(n))$

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 $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n$$

## Using Limits to Compare Order of Growth: Example 1

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

$t(n) = 5n3 + 6n + 2\,, \qquad g(n) = n4$

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

**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

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

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

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

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

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

## 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 ≤ 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
//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 * … *($n$-1) * $n$ for $n \geq 1$ and 0! = 1

Recursive definition of $n$!:

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

$F(0) = 1$

**ALGORITH**M 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
**⇒T(n) ∈ Θ (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) = **2M(n-1) + 1** for n > 0 and M(0)=0
= **$2^n - 1 \in \Theta(2^n)$**
**$\Rightarrow T(n) \in \Theta(2^n)$**

## 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

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

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

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

PES UNIVERSITY

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:

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

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

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

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

…

$= T(n-i) + i$

…

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

## Example2:

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

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

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

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

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

...

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

...

*= 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:

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

$T(1) = 1$

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

$= T(n/2^2) + 1 + 1$

$= T(n/2^3) + 1 + 1 + 1$

.....

$= T(n/2^i) + i$

.....

$= T(n/2^k) + k \quad (k = \log n)$

$= 1 + \log n$

$= O(\log n)$

## Example 4:

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

$T(n) = 2T(n/2) + cn$

$= 2(2T(n/2^2) + c(n/2)) + cn = 2^2 T(n/2^2) + cn + cn$

$= 2^2(2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3 T(n/2^3) + 3cn$

.....

$= 2^i T(n/2^i) + icn$

.....

$= 2^k T(n/2^k) + kcn \quad (k = \log n)$

$= nT(1) + cn \log n = cn + cn \log n$

$= O(n \log n)$

## Example 5:

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

*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*

$S(m)=m+2$

⇨ $T(n)=O(\log n)$

# 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] ǂ K do
        i ← i + 1
    if i < n
        return i
    else
        return -1
```

Basic operation A[i] ǂ K

Basic operation count n (for worst case)

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

## Performance Measurement for sequential search algorithm

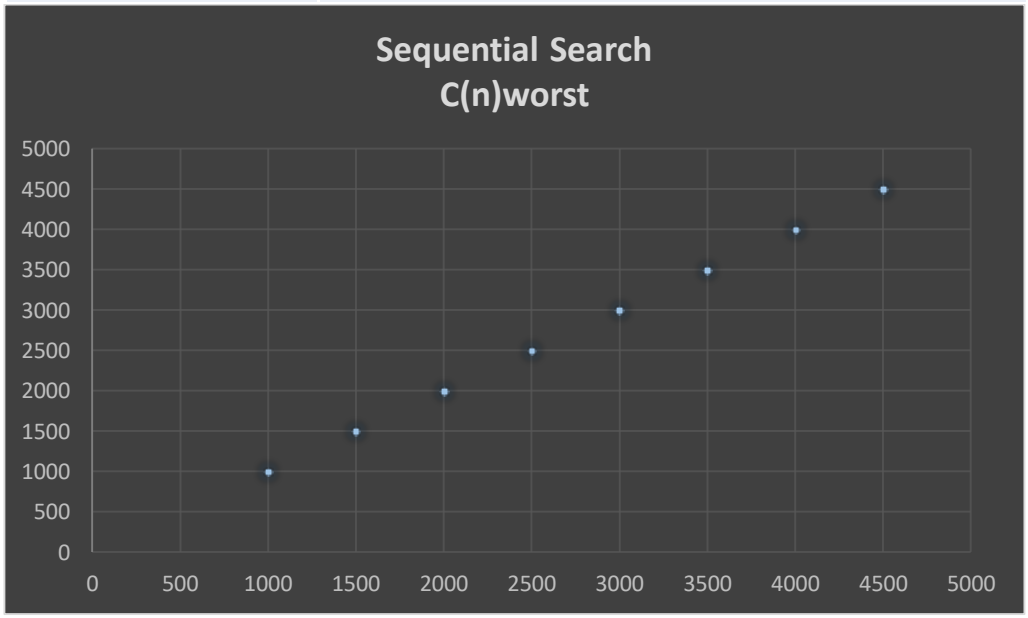
```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
int getrand(int a[], int n)
{
        int i;
         for(i=0;i<n;i++)
         {
                 a[i]=rand()%10000
         }
}
int search(int arr[], int n,int x,int *count)
{
        int i;
        for(i=0;i<n;i++)
        {
     count=count+1;
                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;
```

}

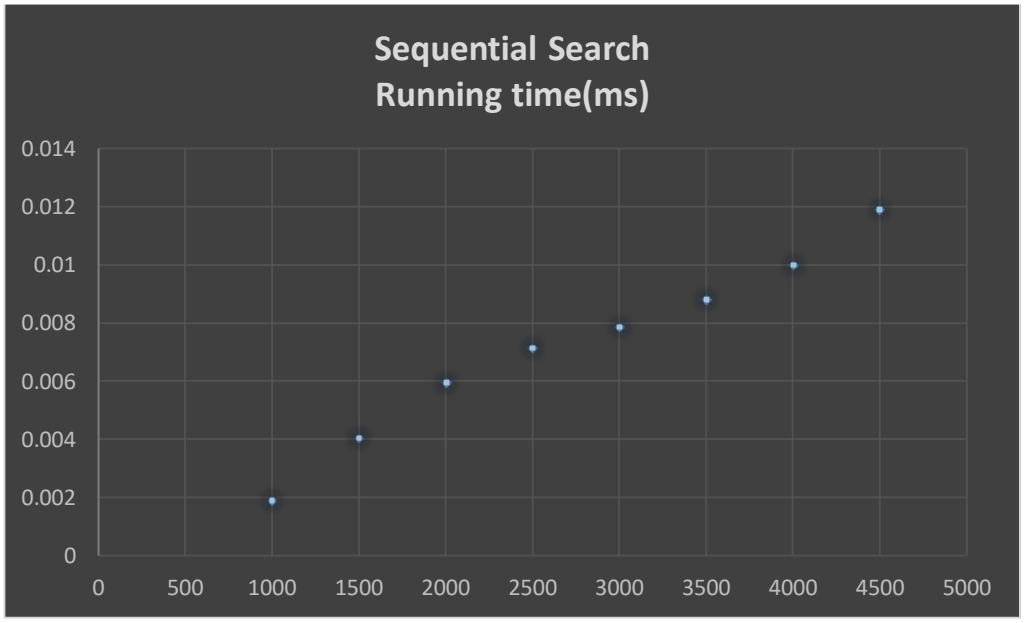
## Number of key comparisons for inputs of different sizes

| Input Size | Sequential Search<br>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<br>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

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Brute Force: Selection Sort

Dr. Shylaja S S

Brute Force: Brute Force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute $a^n$ for a given number **a** and a nonnegative integer **n**. Though this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design techniques, including the brute-force approach. (Also note that computing $a^n$ mod m for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation, $a^n$ = a*…* a. (n times)

This suggests simply computing $a^n$ by multiplying **1** by **a** n times.

*In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of:

- systematically enumerating all possible candidates for the solution
- checking whether each candidate satisfies the problem's statement

A brute-force algorithm to find the divisors of a natural number n would

- enumerate all integers from 1 to n
- check whether each of them divides n without remainder

A brute-force approach for the eight queens puzzle would

- examine all possible arrangements of 8 pieces on the 64-square chessboard
- check whether each (queen) piece can attack any other, for each arrangement

The brute-force method for finding an item in a table (linear search) checks all entries of the table, sequentially, with the item.

*https://en.wikipedia.org/wiki/Brute-force_search

A brute-force search is simple to implement, and will always find a solution if it exists. But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)

Brute-force search is typically used:

- when the problem size is limited
- when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
- when the simplicity of implementation is more important than speed

**Selection Sort**

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n - 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the $i^{th}$ pass through the list, which we number from 0 to n - 2, the algorithm searches for the smallest item among the last n-i elements and swaps it with $A_i$.

$A[0] \leq A[1] \leq A[2] \ldots \leq A[i-1] \mid A[i], \ldots\ldots, A[min], \ldots., A[n-1]$

in their final positions the last n – i elements

After n - 1 passes, the list is sorted.

Here is a pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array.

```
ALGORITHM  SelectionSort(A[0 .. n -1])
//Sorts a given array by selection sort
//Input: An array A[0 .. n - 1]  of orderable elements
//Output: Array A[0 .. n - 1]  sorted in ascending order
for i <- 0 to n - 2 do
        min <- i
        for j <- i+1 to n-1 do
                if A[j] <  A[min]  min  <- j
        swap  A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Fig. 1.

| 89 45 68 90 29 34 **17**
17 | 45 68 90 **29** 34 89
17 29 | 68 90 45 **34** 89
17 29 34 | 90 **45** 68 89
17 29 34 45 | 90 **68** 89
17 29 34 45 68 | 90 **89**
17 29 34 45 68 89 | 90

Fig. 1: Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

The analysis of selection sort is straightforward. The input's size is given by the number of elements n; the algorithm's basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array's size and is given by the following sum:

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

Selection Sort is a $\Theta(n^2)$ algorithm on all inputs.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Bubble Sort

Dr. Shylaja S S

**Bubble Sort**

Bubble Sort is a brute-force application to the sorting problem. In bubble sort, we compare the adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on until, after n - 1 passes, the list is sorted. Pass i ($0 \leq i \leq n-2$) of bubble sort can be represented by the following diagram:

$A[0], A[1], A[2], \ldots, A[j] \overset{?}{\leftrightarrow} A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots. \leq A[n-1]$

in their final positions

Here is a pseudocode of this algorithm.

```
ALGORITHM BubbleSort(A[0 .. n - 1])
//Sorts a given array by bubble sort in their final positions
//Input: An array A[0 .. n - 1] of orderable elements
//Output: Array A[0 .. n- 1] sorted in ascending order
for i <-- 0 to n - 2 do
    for j <-- 0 to n - 2 - i do
        if A[ j + 1 ] < A[ j ] swap A[ j ] and A[ j + 1 ]
```

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Fig. 1.

89 ? 45 68 90 29 34 17
45 89 ? 68 90 29 34 17
45 68 89 ? 90 29 34 17
45 68 89 90 ? 29 34 17
45 68 89 29 90 ? 34 17
45 68 89 29 34 90 ? 17
45 68 89 29 34 17 | 90

45 ? 68 89 29 34 17 | 90
45 68 ? 89 29 34 17 | 90
45 68 89 ? 29 34 17 | 90
45 68 29 89 ? 34 17 | 90
45 68 29 34 89 ? 17 | 90
45 68 29 34 17 | 89 90

Fig. 1: First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

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

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

Bubble Sort is a $\Theta(n^2)$ algorithm.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Sequential Search

Dr. Shylaja S S

**Sequential Search**

The sequential search algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate a check for the list's end on each iteration of the algorithm. Here is a pseudocode for this enhanced version, with its input implemented as an array.

```
AlGORITHM  SequentialSearch2(A[0 .. n ],  K)
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0 .. n -1] whose value is
// equal to K or -1 if no such element is found
A[n]<---K
i<---0
while A[i] ≠K  do
        i<--- i + 1
if i  <  n return i
else return -1
```

Sequential Search is a Θ(n) algorithm.

| 10 | 14 | 19 | 26 | 27 | 31 | 33 | 35 | 42 | 44 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

For key = 33, 6 is returned
For key = 50, -1 is returned (50 is stored in position 10 in the array)

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

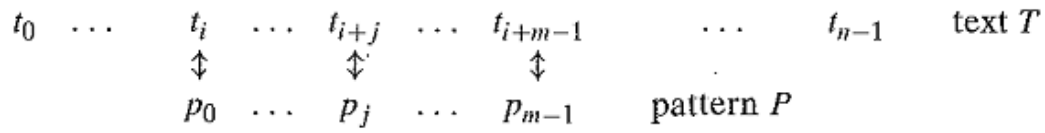
# String Matching

Dr. Shylaja S S

**String Matching**

Given a string of n characters called the **text** and a string of m characters (m ≤n) called the **pattern**; find a substring of the text that matches the pattern. To put it more precisely, we want to find i - the index of the leftmost character of the first matching substring in the text-such that

$t_i = p_0 , \ldots , t_{i+j} = p_j , \ldots . , t_{i+m-1} = p_{m-1}$ :




If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text which can still be a beginning of a matching substring is n - m (provided the text's positions are indexed from 0 to n- 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
ALGORITHM BruteForceStringMatch(T[0 .. n -1], P[0 .. m -1])
//Implements brute-force string matching
//Input: An array T[0 .. n - 1] of n characters representing a text and an array
//P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a matching
//substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
        j ←0
        while j < m and P[j] = T[i + j] do
                j ← j+1
        if j = m return i
return -1
```

**Example:**

N 0 B 0 D Y _ N 0 T I C E D _ H I M

**N** 0 T

  **N** 0 T

    **N** 0 T

      **N** 0 T

        **N** 0 T

          **N** 0 T

            **N** 0 T

              **N 0 T**

Fig 1: Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. However, the worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n-m+1 tries. Thus, in the worst case, the algorithm is in Θ(nm). For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., Θ(n + m) = Θ(n).

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Exhaustive Search: Travelling Salesman Problem

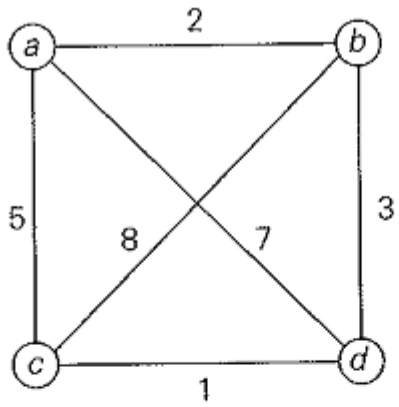
Dr. Shylaja S S

Many important problems require finding an element with a special property in a domain that grows exponentially (or faster) with an instance size. Typically, such problems arise in situations that involve-explicitly or implicitly-combinatorial objects such as permutations, combinations, and subsets of a given set. Many such problems are optimization problems: they ask to find an element that maximizes or minimizes some desired characteristic such as a path's length or an assignment's cost.

**Exhaustive search** is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem's domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that though the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects. We assume here that they exist. We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem. In this section, we shall discuss the traveling salesman problem.

The traveling salesman problem (TSP) has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805-1865), who became interested in such cycles as an application of his algebraic discoveries.) It is easy to see that a Hamiltonian circuit can be also defined as a sequence of $n + 1$ adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one while all the other $n - 1$ vertices are distinct. Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex (they are cycles after all, are they

not?). Thus, we can get all the tours by generating all the permutations of n - 1 intermediate cities, compute the tour lengths, and find the shortest among them. Fig. 1 presents a small instance of the problem and its solution by this method.




| Tour | Length | |
|---|---|---|
| $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$ | $l = 2 + 8 + 1 + 7 = 18$ | |
| $a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$ | $l = 2 + 3 + 1 + 5 = 11$ | optimal |
| $a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$ | $l = 5 + 8 + 3 + 7 = 23$ | |
| $a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$ | $l = 5 + 1 + 3 + 2 = 11$ | optimal |
| $a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$ | $l = 7 + 3 + 8 + 5 = 23$ | |
| $a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$ | $l = 7 + 1 + 8 + 2 = 18$ | |

Fig. 1: Solution to a small instance of the traveling salesman problem by exhaustive search

An inspection of Fig. 1 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, B and C, and then consider only permutations in which B precedes C. (This trick implicitly defines a tour's direction.)

This improvement cannot brighten the efficiency picture much, however. The

total number of permutations needed will still be (n- 1)!/2, which makes the exhaustive-search approach impractical for all but very small values of n. On the other hand, if you always see your glass as half-full, you can claim that cutting the work by half is nothing to sneeze at, even if you solve a small instance of the problem, especially by hand. Also note that had we not limited our investigation to the circuits starting at the same vertex, the number of permutations would have been even larger, by a factor of n.

The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other n – 1 cities. Thus, the total number of permutations needed will be (n – 1)!

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

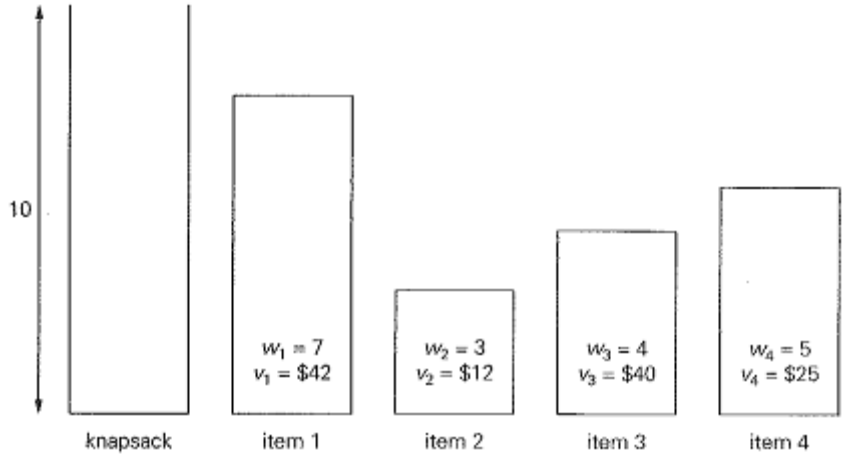
UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Exhaustive Search: Knapsack Problem

Dr. Shylaja S S

**Knapsack Problem**

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity. Fig. 1 presents a small instance of the knapsack problem.




(a)

| Subset | Total weight | Total value |
|---|---|---|
| ∅ | 0 | $0 |
| {1} | 7 | $42 |
| {2} | 3 | $12 |
| {3} | 4 | $40 |
| {4} | 5 | $25 |
| {1, 2} | 10 | $36 |
| {1, 3} | 11 | not feasible |
| {1, 4} | 12 | not feasible |
| {2, 3} | 7 | $52 |
| {2, 4} | 8 | $37 |
| **{3, 4}** | **9** | **$65** |
| {1, 2, 3} | 14 | not feasible |
| {1, 2, 4} | 15 | not feasible |
| {1, 3, 4} | 16 | not feasible |
| {2, 3, 4} | 12 | not feasible |
| {1, 2, 3, 4} | 19 | not feasible |

(b)

Fig. 1: (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack's capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Fig. 1a is given in Fig. 1b. Since the number of subsets of an n-element set is $2^n$, the exhaustive search leads to a $\Omega(2^n)$ algorithm no matter how efficiently individual subsets are generated.

Thus, for both the traveling salesman and knapsack problems, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so-called NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven. More sophisticated approaches - backtracking and branch-and-bound enable us to solve some but not all instances of these (and similar) problems in less than exponential time. Alternatively, we can use one of many approximation algorithms.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

# Exhaustive Search: Assignment Problem

Dr. Shylaja S S

**Assignment Problem**

There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the $i^{th}$ person is assigned to the $j^{th}$ job is a known quantity C[i, j] for each pair i, j = 1, 2, ... , n. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs C[i, j]:

| | Job 1 | Job 2 | Job 3 | Job 4 |
|---|---|---|---|---|
| Person 1 | 9 | 2 | 7 | 8 |
| Person 2 | 6 | 4 | 3 | 7 |
| Person 3 | 5 | 8 | 1 | 8 |
| Person 4 | 7 | 6 | 9 | 4 |

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix C. In terms of this matrix, the problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible. Note that no obvious strategy for finding a solution works here. For example, we cannot select the smallest element in each row because the smallest elements may happen to be in the same column. In fact, the smallest element in the entire matrix need not be a component of an optimal solution. Thus, opting for the exhaustive search may appear as an unavoidable evil.

We can describe feasible solutions to the assignment problem as n-tuples $<j_1, \ldots, j_n>$ in which the $i^{th}$ component, i = 1, ... , n, indicates the column of the element selected in the $i^{th}$ row (i.e., the job number assigned to the $i^{th}$ person). For example, for the cost matrix above, <2, 3, 4, 1> indicates a feasible assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and

permutations of the first n integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, ..., n, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are shown in Fig. 1; you may complete the remaining.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23
etc.

Fig. 1: First few iterations of solving a small instance of the assignment problem by exhaustive search

Since the number of permutations to be considered for the general case of the assignment problem is n!, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the Hungarian method after the Hungarian mathematicians Konig and Egervary whose work underlies the method.