Why study algorithms?

- Theoretical importance
  - the core of computer science

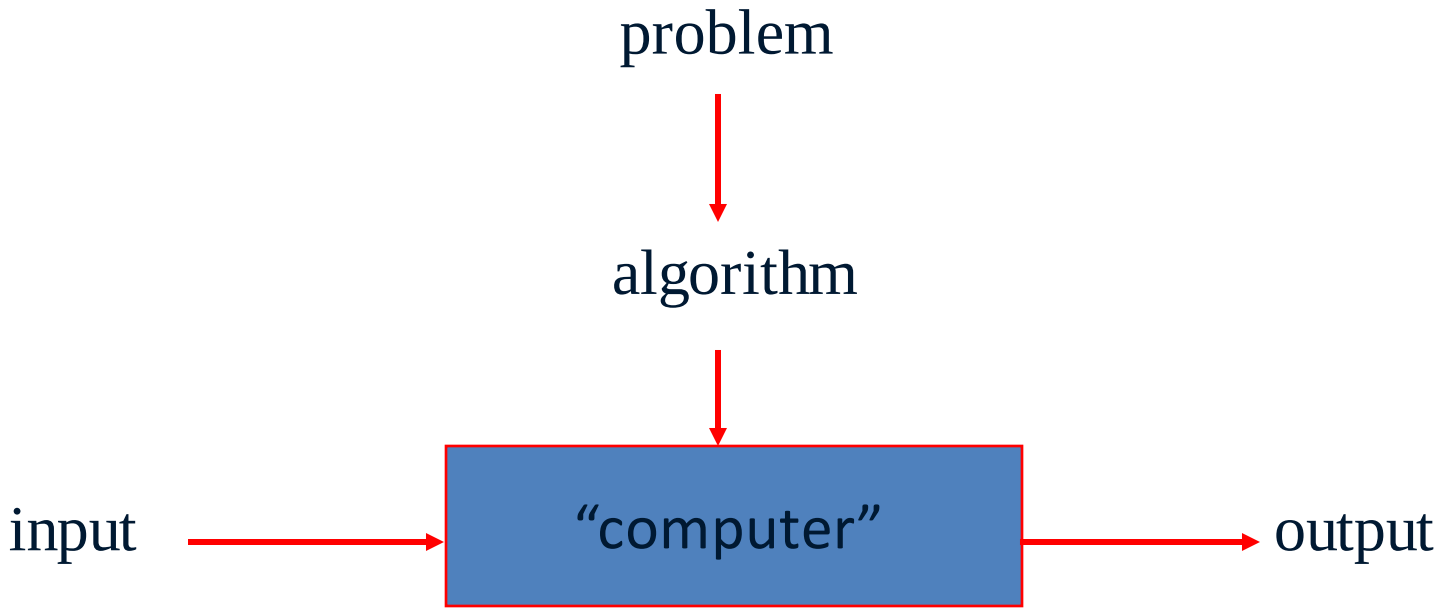
- Practical importance
  - Framework for designing and analyzing algorithms for new problems

PageRank is a way of measuring the importance of website pages

Example: Google's PageRank Technology

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.



Characteristics of Algorithm

- 1) Must take an input
- 2)Must give some output
- 3)Definiteness: Each step must be clear and unambiguous
- 4)Effectiveness: Every instruction must be basic
  i. e simple instruction
- 5)Finiteness: Algorithm terminates after finite number of steps
- 6)correctness: correct set of values must be produce from the each set of inputs

Euclid's Algorithm

Problem: Find gcd(*m,n*), the greatest common divisor of two nonnegative, not both zero integers *m* and *n*

Euclid's algorithm is based on repeated application of equality

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

until the second number becomes 0

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Two descriptions of Euclid's algorithm

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.

Algorithm Euclid(m,n)
//Input:Two integers m and n
//Output:Greatest common divisor of m and n
while $n \neq 0$ do
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

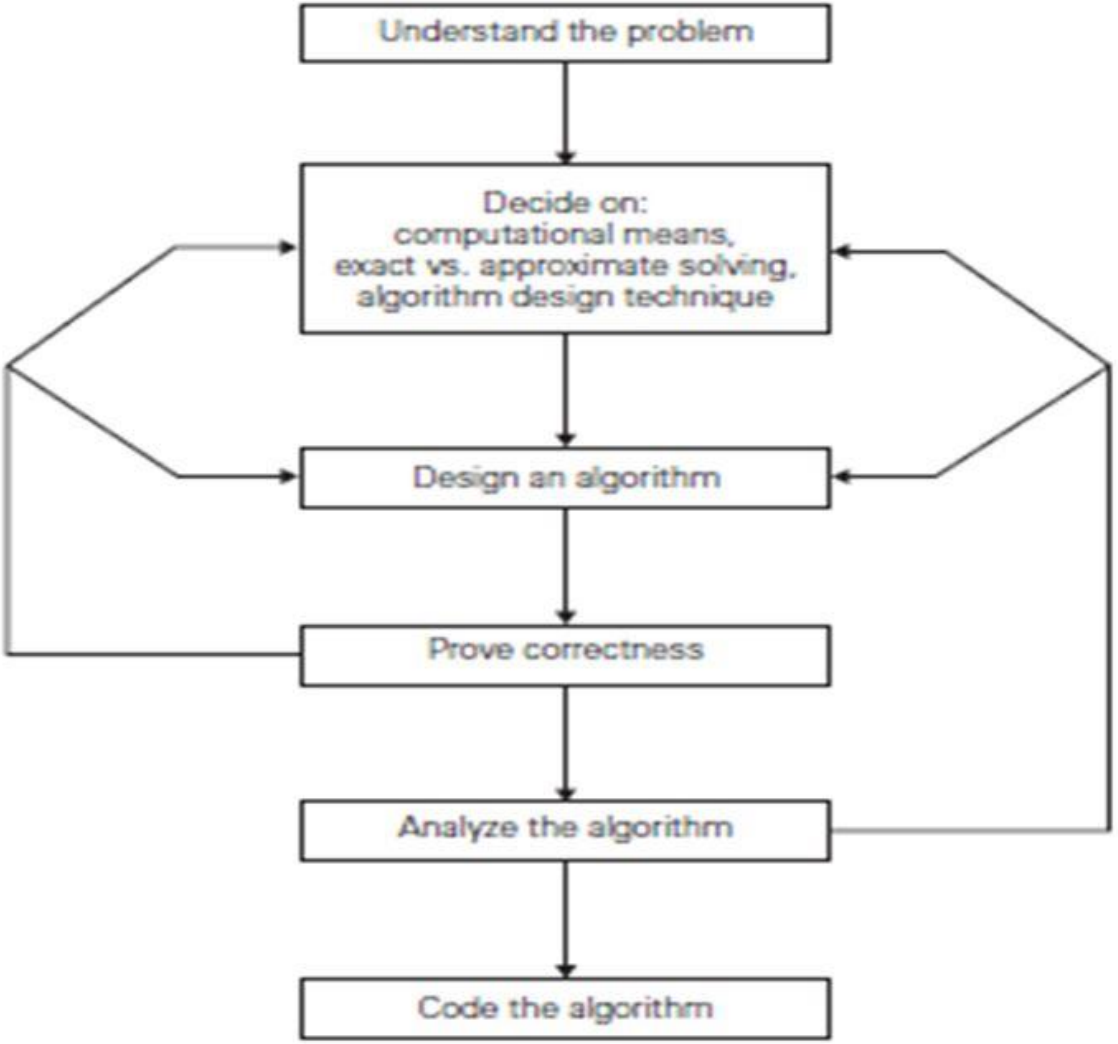
Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

Problem Solving

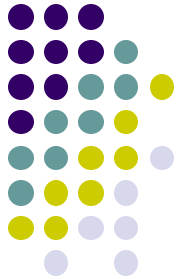
- Programming is a process of problem solving
- Problem solving techniques
  - Analyze the problem
  - Outline the problem requirements
  - Design steps (algorithm) to solve the problem
- Algorithm:
  - Step-by-step problem-solving process
  - Solution achieved in finite amount of time

Fundamentals of Algorithm and Problem Solving



Important Problem Types

- Sorting
- Searching
- String processing (e.g. string matching)
- Graph problems (e.g. graph coloring problem)
- Combinatorial problems (e.g. maximizes a cost)
- Geometric problems (e.g. convex hull problem)
- Numerical problems (e.g. solving equations)

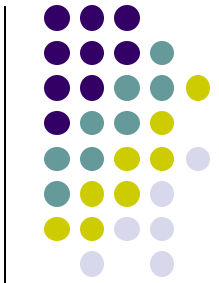


Fundamentals of the Analysis of algorithm efficiency

- There are two kinds of efficiencies to analyze the algorithm
- Time efficiency
- Space efficiency

- The algorithm analysis framework consists of
- Measuring an input size
- Unit for Measuring running time
- Orders of growth
- Best case worst case and avg case efficiencies

Space efficiency



```
Add()
{
int z=a+b+c;
return z
}
```

Variables a,b,c,z are integers hence they require 4 bytes each total memory required is 16bytes

S(p) =C+sp

C constant space taken by instruction variable and identifier

sp space dependent upon instance characteristics (referenced variable, recursion stack space)

Space efficiency

```
 sum (  a, n)
{
int s=0;
for(i=0;i<n;i++)
{
 s=+a[i];
}
return (s);
}
```

n-4 bytes

a-4*n bytes

i-4 bytes s-4 bytes

4n+12

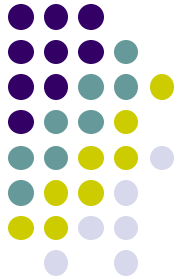
Time Complexity

```
sum (  a, n)
{
int s=0;
for(i=0;i<n;i++)
{
 s=+a[i];
}
return (s);
}
```

- Time complexity is 2n+2

Tabular Method

***Figure 1.2: Step count table for Program 1.10 (p.26)**

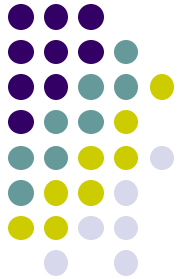


Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Measuring an input size



- Efficiency is also based on input size
- Input size depends on the problem
- Example
- What is input size for sorting n numbers
- What is the input size for polynomial
- What is the input size for multilying two matrices

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes the most towards the running time of the algorithm

$$T(n) \approx c_{op}C(n)$$

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	**Number of list's items, i.e. *n***	**Key comparison**
Multiplication of two matrices	**Matrix dimensions or total number of elements**	**Multiplication of two numbers**
Checking primality of a given integer *n*	***n*'size = number of digits (in binary representation)**	**Division**
Typical graph problem	**#vertices and/or edges**	**Visiting a vertex or traversing an edge**

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n

Example: 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: 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
```

- Worst case O(n) n key comparisons
- Best case O(1) 1 comparisons
- Average case (n+1)/2 (n+1)/2, assuming K is in A

Order of growth

- Order of growth in algorithm means how the time for computation increases when you increase the input size. It really matters when your input size is very large.

Order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem

- Depending on the algorithm, the behavior changes.

Values of some important functions as $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms