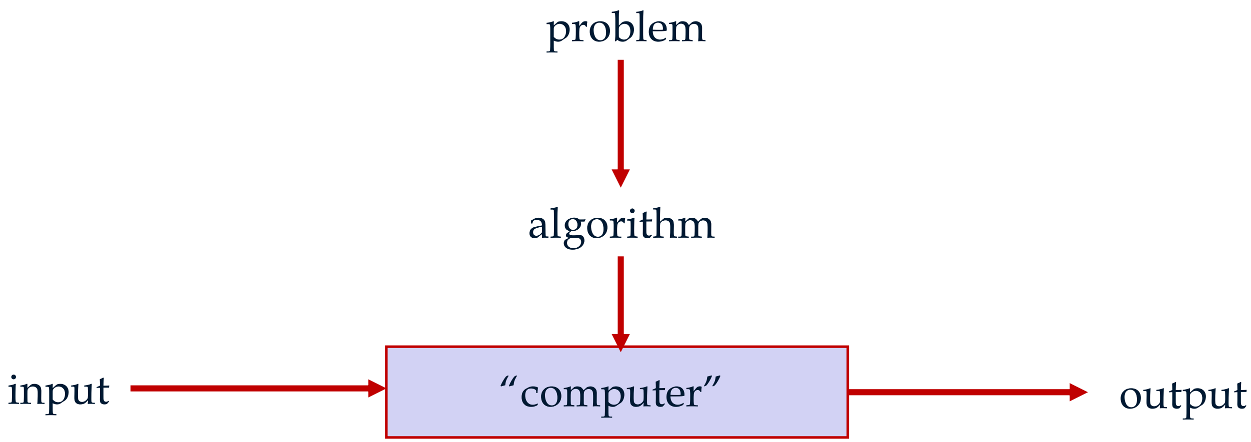
Algorithms and Their Applications
- Fundamentals of Arithmetic Problem Solving -

Won-Yong Shin

March 16th, 2020

연세대학교
YONSEI UNIVERSITY

- Algorithm
  - 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



Methods for **gcd**(m, n): Middle-School Version

- Middle-school procedure
  - Step 1: Find the prime factorization of *m*
  - Step 2: Find the prime factorization of *n*
  - Step 3: Find all the common prime factors
  - Step 4: Compute the product of all the common prime factors and return it as **gcd**(*m*,*n*)

  - Example: **gcd**(60, 24)

Euclid's algorithm

- Based on repeated application of equality **gcd**(m,n) = **gcd**(n, m mod n) until the second number becomes 0, which makes the problem trivial

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

- 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.

```
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Pseudocode

Methods for **gcd**(m, n): Consecutive Integer Checking Algorithm

- Consecutive integer checking algorithm
  - Step 1: Assign the value of min{m,n} to t
  - Step 2: Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4
  - Step 3: Divide n by t. If the remainder is 0, **return t** and stop; otherwise, go to Step 4
  - Step 4: Decrease t by 1 and go to Step 2

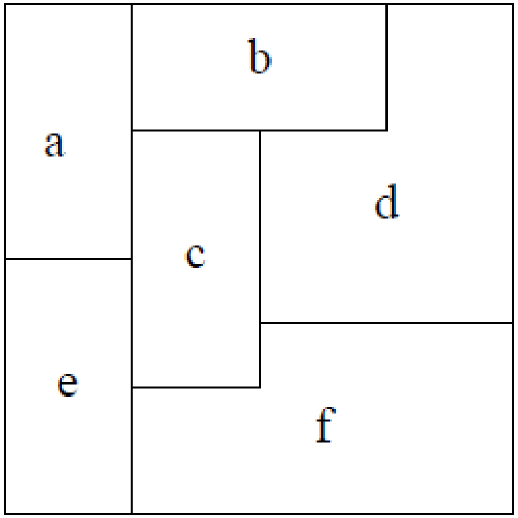
- Algorithms (by Anany Levitin)
  - Brute force
  - Divide and conquer
  - Decrease and conquer
  - Transform and conquer
    - Heaps and heapsort
  - Space and time tradeoffs
  - Dynamic programming
  - Greedy approach
  - Iterative improvement
  - Backtracking
  - Branch and bound

- How good is the algorithm?
  - *Time* efficiency
  - *Space* efficiency
  - Example: algorithms for **gcd**(*m*,*n*)

- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

- Sorting
- Searching
- String processing
  - cf. string matching
- Graph problems
  - Traveling salesman problem: finding the shortest tour through n cities that visits every city exactly once
  - Graph-coloring problem: assigning the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color
- Combinatorial problems
- Geometric problems
  - Closest-pair problem: finding the closest pair among given n points
  - Convex-hull problem: Finding the smallest convex polygon that would include all the points of a given set
- Numerical problems

Consider the following map:



a. Explain how we can use the graph-coloring problem to color the map so that no two neighboring regions are colored the same.

b. Use your answer to part (a) to color the map with the smallest number of colors.

Theoretical Analysis of Time Efficiency

- **Time** efficiency
  - Analyzed by determining the number of repetitions of the basic operation as a function of input size
    - *Basic operation*: the operation that contributes most towards the running time of the algorithm

Input size

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

Running time

Execution time for basic operation

Number of times basic operation is executed

Examples: Input Size and Basic Operation

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	Size of n = number of digits (in binary representation)	Division
Typical graph problem	# of vertices and/or edges	Visiting a vertex or traversing an edge

Types of Formulas for Basic Operation's Count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with *specific* multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with *unknown* multiplicative constant

  e.g., $C(n) \approx cn^2$

- Most important: *order of growth* within a constant multiple as $n \to \infty$
- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

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

- 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
    - Number of times the basic operation will be executed on typical input
    - NOT the average of worst and best case
    - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

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
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

- Comparison
  - Worst-case
  - Best-case
  - Average-case