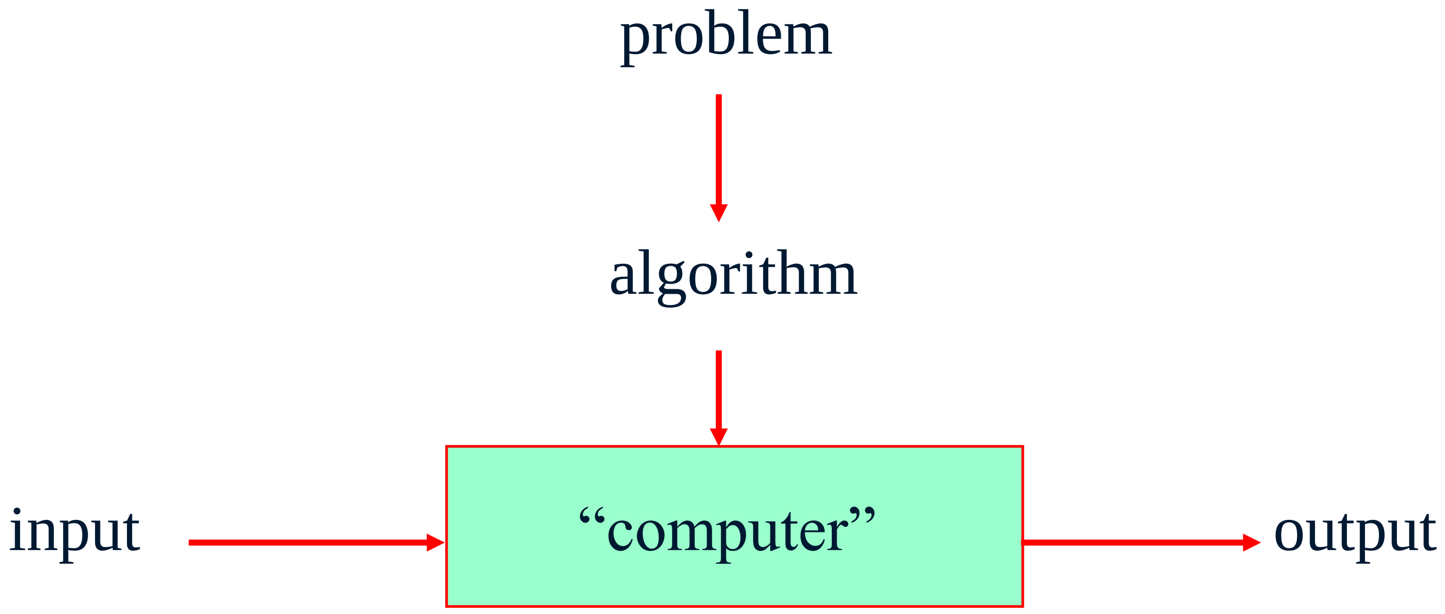
CSE408 Fundamentals of Algorithms

Lecture #1

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.



Algorithm

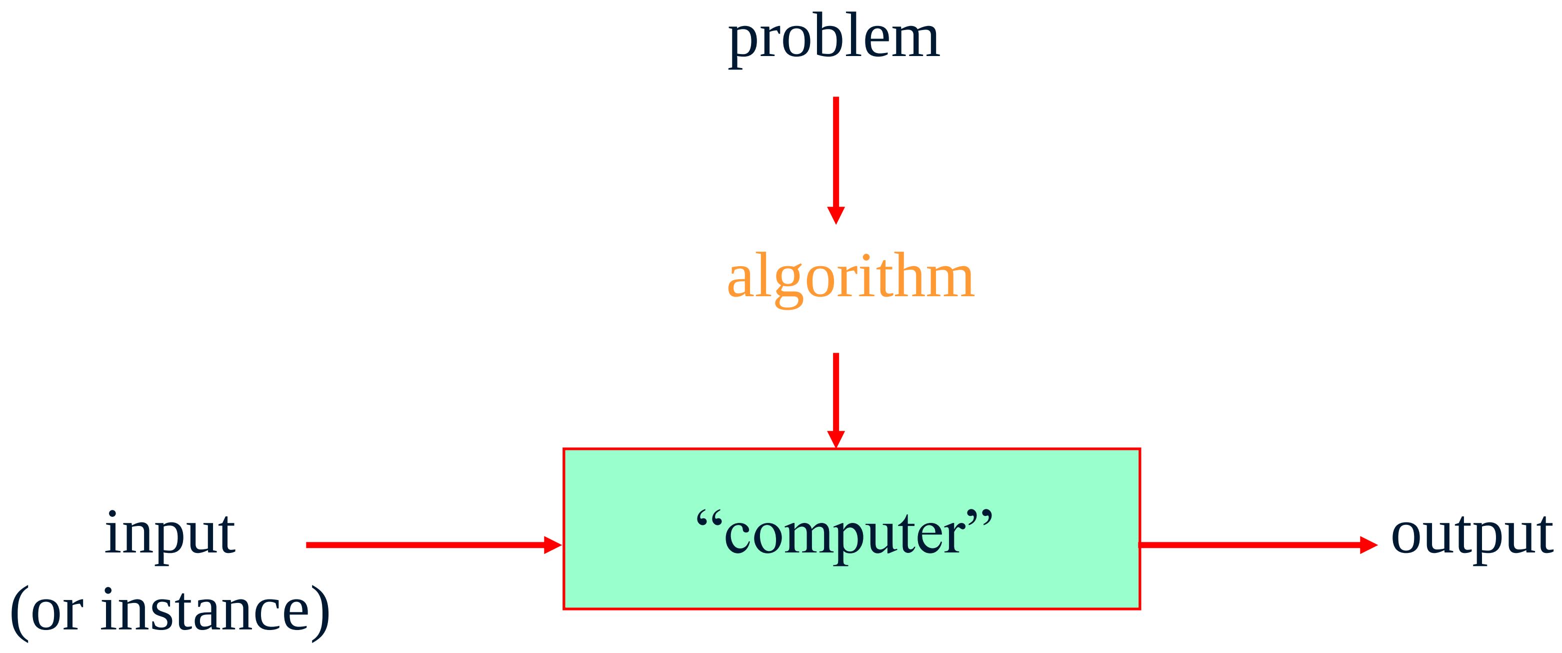
- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Historical Perspective

- Euclid’s algorithm for finding the greatest common divisor

- Muhammad ibn Musa al-Khwarizmi – 9th century mathematician
  www.lib.virginia.edu/science/parshall/khwariz.html

Notion of algorithm and problem



(different from a conventional solution)

Example of computational problem: sorting

- Statement of problem:
  - *Input:* A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - *Output:* A reordering of the input sequence $<a'_1, a'_2, \ldots, a'_n>$ so that $a'_i \leq a'_j$ whenever $i < j$
- Instance: The sequence <5, 3, 2, 8, 3>
- Algorithms:
  - Selection sort
  - Insertion sort
  - Merge sort
  - (many others)

Selection Sort

- Input: array $a[1], \cdots, a[n]$
- Output: array a sorted in non-decreasing order
- Algorithm:

```
for $i$=1 to $n$
    swap $a[i]$ with smallest of $a[i], \cdots, a[n]$
```

Is this unambiguous? Effective?

Some Well-known Computational Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency (or complexity) analysis
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and branch-and-bound
- Space and time tradeoffs

Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency

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

What is an algorithm?

- Recipe, process, method, technique, procedure, routine,… with the following requirements:

1. Finiteness
    - terminates after a finite number of steps
2. Definiteness
    - rigorously and unambiguously specified
3. Clearly specified input
    - valid inputs are clearly specified
4. Clearly specified/expected output
    - can be proved to produce the correct output given a valid input
5. Effectiveness
    - steps are sufficiently simple and basic

Why study algorithms?

- Theoretical importance
  - the core of computer science
- Practical importance
  - A practitioner’s toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Example: Google’s PageRank Technology

Euclid’s Algorithm

As per Euclid's algorithm for the greatest common divisor, the GCD of two positive integers (a, b) can be calculated as:

- If a = 0, then GCD (a, b) = b as GCD (0, b) = b.
- If b = 0, then GCD (a, b) = a as GCD (a, 0) = a.
- If both a≠0 and b≠0, we write 'a' in quotient remainder form (a = b×q + r) where q is the quotient and r is the remainder, and a>b.
- Find the GCD (b, r) as GCD (b, r) = GCD (a, b)
- We repeat this process until we get the remainder as 0.

- For a set of two positive integers (a, b) we use the below-given steps to find the greatest common divisor:
- **Step 1:** Write the divisors of positive integer "a".
- **Step 2:** Write the divisors of positive integer "b".
- **Step 3:** Enlist the common divisors of "a" and "b".
- **Step 4:** Now find the divisor which is the highest of both "a" and "b".

Example: Find the greatest common divisor of 13 and 48.

Solution: We will use the below steps to determine the greatest common divisor of (13, 48).

Divisors of 13 are 1, and 13.

Divisors of 48 are 1, 2, 3, 4, 6, 8, 12, 16, 24 and 48.

The common divisor of 13 and 48 is 1.

The greatest common divisor of 13 and 48 is 1.

Thus, GCD(13, 48) = 1.

GCD using LCM Method

- The steps to calculate the GCD of (a, b) using the LCM method is:
- **Step 1:** Find the product of a and b.
- **Step 2:** Find the least common multiple (LCM) of a and b.
- **Step 3:** Divide the values obtained in Step 1 and Step 2.
- **Step 4:** The obtained value after division is the greatest common divisor of (a, b).

$$\boxed{\text{GCD}(a, b) = \frac{(a \times b)}{\text{LCM}(a, b)}}$$

Example: Find the greatest common divisor of 15 and 70 using the LCM method.

Solution: The greatest common divisor of 15 and 70 can be calculated as:

The product of 15 and 70 is given as, 15 × 70

The LCM of (15, 70) is 210.

GCD (15, 20) = (15 × 70)/ 210 = 5.

∴ The greatest common divisor of (15, 70) is 5.

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.

```
while $n \neq 0$ do
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return $m$
```

Other methods for computing gcd(*m*,*n*)

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
```

Is this slower than Euclid's algorithm?
How much slower?

$O(n)$, if $n <= m$, vs $O(\log n)$

Other methods for gcd(*m*,*n*) [cont.]

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*)

Is this an algorithm?

How efficient is it?

Time complexity: O(sqrt(n))

Sieve of Eratosthenes

Input: Integer $n \geq 2$

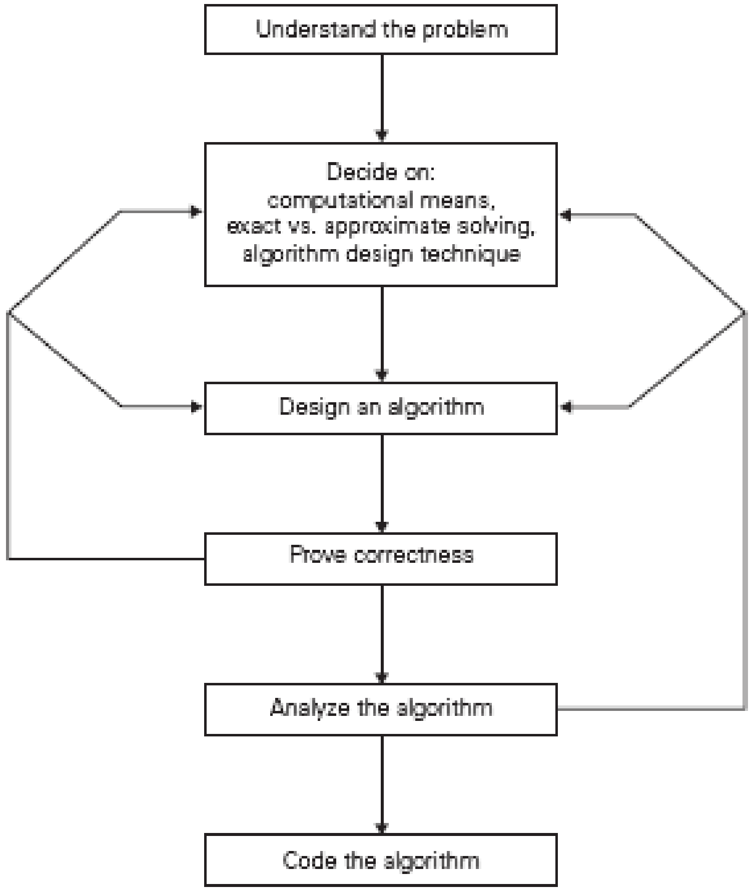
Output: List of primes less than or equal to n

```
for $p \leftarrow 2$ to $n$ do  $A[p] \leftarrow p$
for $p \leftarrow 2$ to $n$ do
    if $A[p] \neq 0$  //$p$ hasn't been previously eliminated from the list
      $j \leftarrow p * p$
      while $j \leq n$  do
          $A[j] \leftarrow 0$  //mark element as eliminated
          $j \leftarrow j + p$
```

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Time complexity: O(n)

Fundamentals of Algorithmic Problem Solving



Two main issues related to algorithms

- How to design algorithms

- How to analyze algorithm efficiency

Algorithm design techniques/strategies

The algorithms can be classified in various ways. They are:

- Implementation Method
- Design Method
- Design Approaches
- Other Classifications

Implementation Method

- **Recursion or Iteration:** A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.
  Example: The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.

- **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.
  Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

- **Serial or Parallel or Distributed Algorithms:** In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

Design Method

- **Greedy Method:** In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.
  Example: Fractional Knapsack, Activity Selection.
- **Divide and Conquer:** The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.
  Example: Merge sort, Quicksort.

- **Dynamic Programming:** The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.
  Example: 0-1 Knapsack, subset-sum problem.
- **Linear Programming:** In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.
  Example: Maximum flow of Directed Graph

- **Reduction(Transform and Conquer):** In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.
   Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer*.

- **Backtracking:** This technique is very useful in solving combinatorial problems that have *a single unique solution*. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn’t seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.
  Example: N-queen problem, maize problem.

- **Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution.
  Example: Job sequencing, Travelling salesman problem.

Design Approaches

- **Top-Down Approach:** In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.
- **Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches.
  In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.

Other Classifications

- **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.
  Example: Randomized Quicksort Algorithm.
- **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.
  Example: Some algorithms take O(n), while some take exponential time.
- **Classification by Research Area:** In CS each field has its own problems and needs efficient algorithms.
  Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.
- **Branch and Bound Enumeration and Backtracking:** These are mostly used in Artificial Intelligence.

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
  - correctness ignored in this course

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Important problem types

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

Sorting (I)

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key
  - A specially chosen piece of information used to guide sorting. E.g., sort student records by names.

Sorting (II)

- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - 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.

Selection Sort

```
Algorithm SelectionSort(A[0..n-1])
//The algorithm 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]
```

Searching

- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  - Sequential search
  - Binary search …

Time: O(log n)

String Processing

- 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.

Thank You !!!