Summary

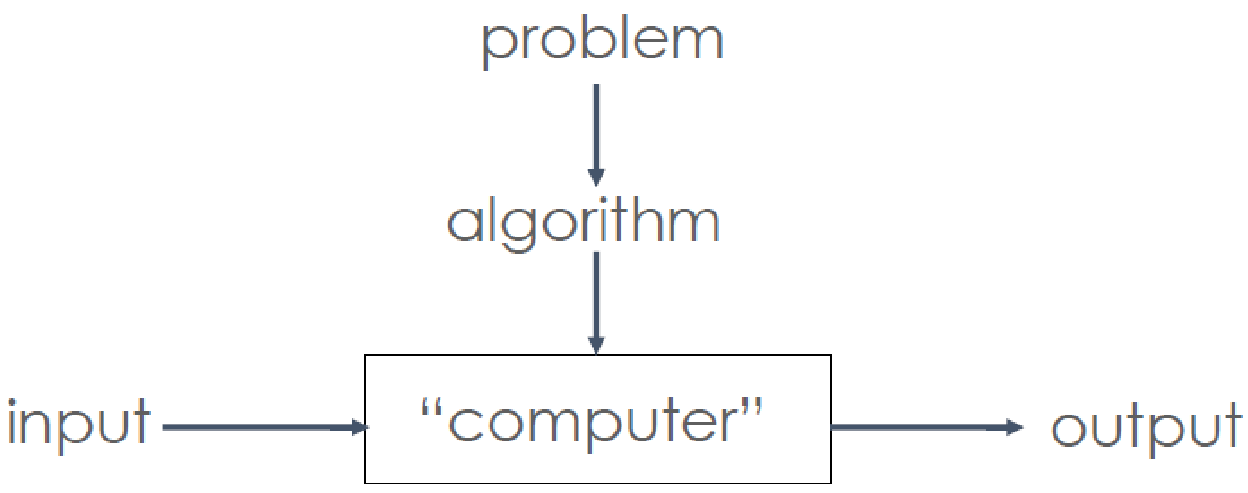
What is an algorithm?

- A sequence of unambiguous instructions for solving a well-defined problem. Algorithms are guaranteed to terminate if the input is valid.
- Algorithms are a subset of procedures, which aren't guaranteed to terminate.

Terms

- Finite
  - Terminates after a finite number of steps
- Definite
  - Rigorously and unambiguously specified
- Inputs
  - Valid inputs are clearly specified
- Output
  - Can be proved to produce the correct output given a valid input
- Effective
  - Steps are sufficiently simple and basic

Notion of an Algorithm

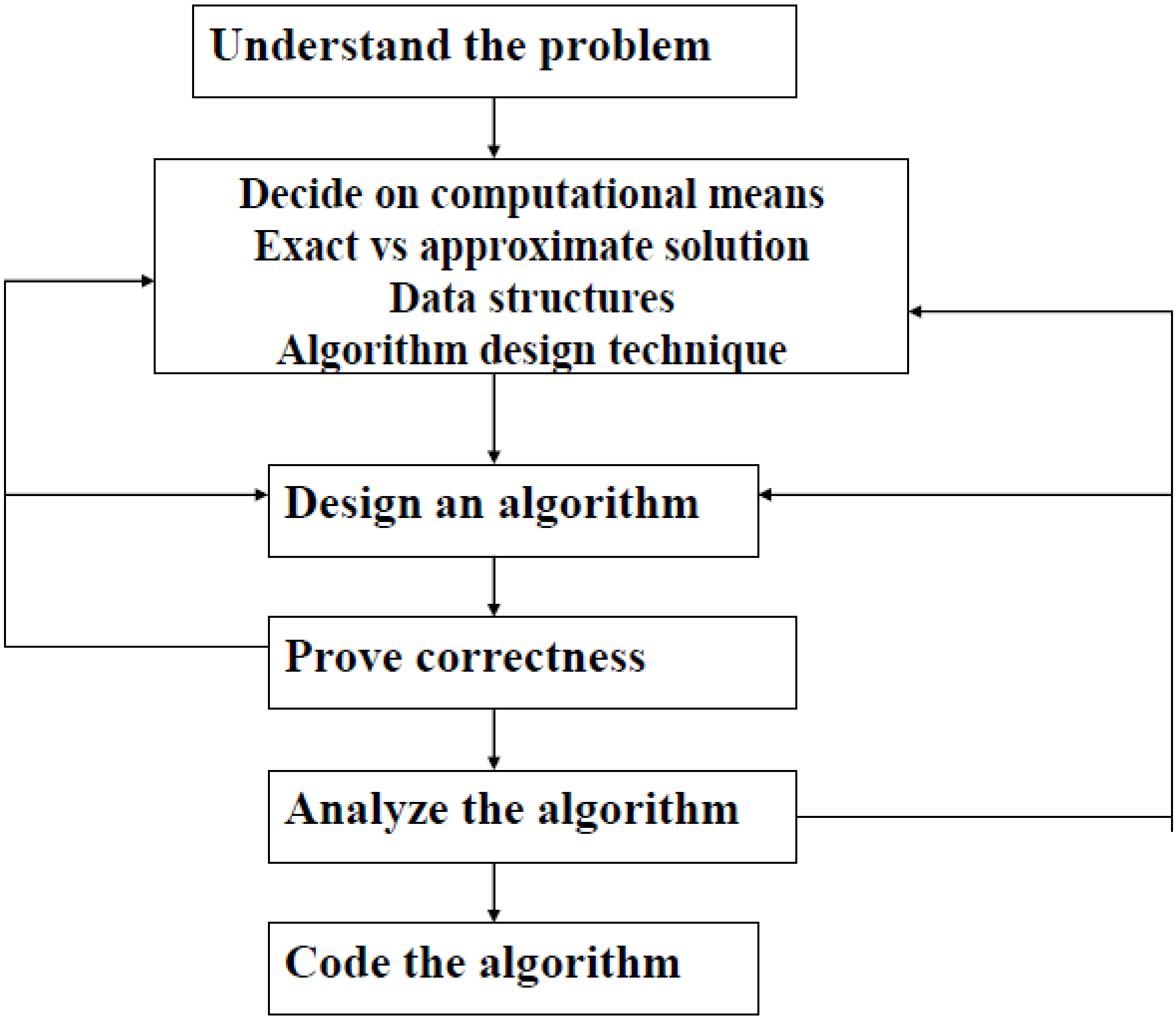


- Each step of the algorithm must be **unambiguous**
- The **range of inputs** must be specified carefully.
- The same algorithm can be represented in different ways AND Several algorithms for solving the same problem may exist - with different properties

Methodology of Algorithms

1. Understand the problem
2. Decide on computational means
3. Design an algorithm
4. Prove correctness
5. Analyze the algorithm
6. Code algorithm

Another representation of the above 6 steps:



Analyzing Algorithms

- Efficiency: time and space
- Simplicity
- Generality: range of inputs, special cases
- Optimality: no other algorithm can do better

Types of Algorithms

- Brute Force
  - Try all possibilities
- Decrease & Conquer
  - Solve large instance in terms of smaller instance

- Divide & Conquer
    - Break problem into distinct subproblems
- Transform & Conquer
    - AKA Transformation
    - Covert problem to another one
- Trading Space & Time
    - Use additional data structures
- Dynamic Programming
    - Break problem into overlapping subproblems
- Greedy
    - Repeatedly do what is best now

Formal Definitions - O, Ω & Θ

- Definition: *f(n) ∈ O(g(n))* iff there exists positive constant c and non-negative integer n_0 such that ** f(n) <= c g(n)* for every n >=n_0

- Definition: *f(n) ∈ Ω(g(n))* iff there exist positive constant c and non-negative integer n_0 such that

    - *f(n)* >= *c g(n)* for every n >= n_0

- Definition: *f(n) ∈ Θ(g(n))* iff there exist positive constants c_1 and c_2 and non-negative integer n_0 and non-negative integer n_0 such that

    - c_1 *g(n)* <= *f(n)* <= c_2 *g(n)* for every n >= n_0

- O(g(n)): functions that grow no faster than g(n)

- Ω(g(n)): functions that grow at least as fast as g(n)

- Θ(g(n)): functions that grow at same rate as g(n)

- O(g(n)): functions no worse than g(n)

- Ω(g(n)): functions at least as bad as g(n)

- Θ(g(n)): functions as efficient as g(n)

O vs Θ

- O is an upper bound on performance
- Θ is a tight bound
    - It is the upper and lower bound

Brute Force

A straightforward approach usually directly based on problem statement and definitions

- Crude but often effective

- Simple
- Widely Applicable
- Sometimes impractically slow
- Try all the possibilities until problem solved
- Loop through each possibility, check if it solves problems

Pros and Cons of Brute Force

- Strengths
    - Wide applicability
    - Simplicity
    - Yields reasonable algorithm for some important problems and standard algorithms for simple computational tasks
    - A good yardstick for better algorithms
    - Sometimes doing better is not worth the bother
- Weakness
    - Rarely produces efficient algorithms
    - Some brute force algorithms are infeasibly slow
    - Note as creative as some other design techniques

Exhaustive Search

- Definition
    - A brute force solution to the search for an element with a special property
    - Usually among combinatorial objects such a permutations or subsets
    - Suggests generating each and every element of the problem's domain
- Method
    1. Construct a way of listing all potential solutions to the problem in a systematic manner
    2. Evaluate all Solutions one by one (disqualifying infeasible ones) keeping track of the best one found so far
    3. When search ends, announce the winner

Comments on Exhaustive Search

- Exhaustive search algorithms run in a realistic amount of time **only on very small instances**
- In many cases there are much better alternatives!
- In some cases exhaustive search (or variation) is the only known solution
- and parallel solutions can speed it up

Master Theorem

If we have a recurrence of this form:

$T(n) = aT(n/b) + f(n)$ and $f(n) \in \Theta\ (n^2)$ then:

- $T(n) \in \Theta(n^d)$ if $a < b^d$

- $T(n) \in \Theta(n^d \log n)$ if $a = b^d$
- $T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$

Divide & Conquer

- Divide & Conquer is the best known algorithm design strategy:
  1. Divide instances of problem into two or more smaller instances
  2. Solve smaller instances recursively
  3. Obtain solution to original (larger) instance by combining these solutions

Decrease & Conquer

- Decrease by a constant
- Decrease by a constant factor
- Variable size decrease

Decrease by a Constant Factor

1. Reduce problem instance to smaller instance of the same problem and extend solution
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original problem
   - Also called inductive or incremental

Strengths and Weaknesses of Decrease & Conquer

Strengths

- Can be implemented either top down (recursively) or bottom up (without recursion)
- Often very efficient (possibly $\Theta(\log n)$)
- Leads to a powerful form of graph traversal (breadth and depth first search)

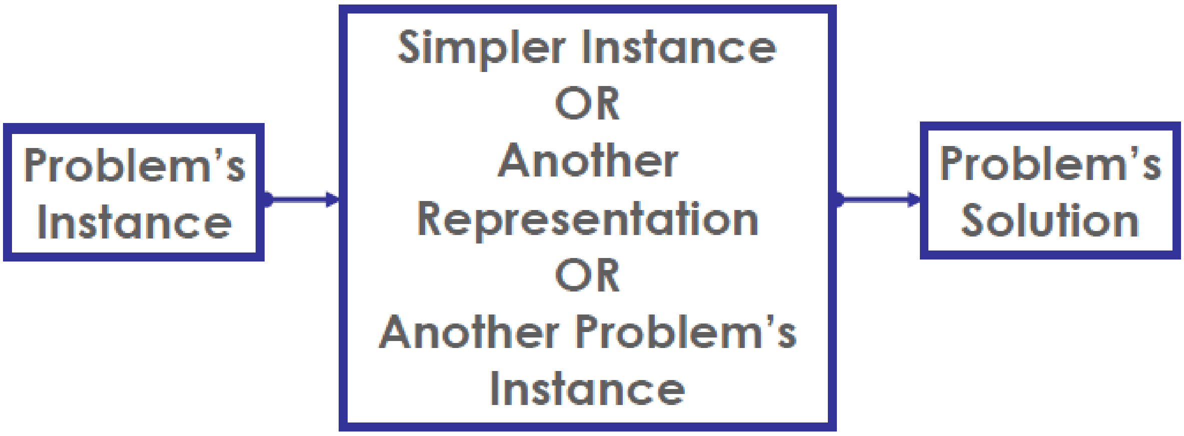
Weakness

- Less widely applicable (especially decrease by a constant factor)

Transform and Conquer

> The secret of life is to replace one worry with another

👆 Charles M. Schultz



Different types of transformations:

1. Instance simplification = a more convenient instance of the same problem
    - Presorting
    - Gaussian elimination
2. Representation Change = a different representation of the same instance
    - Balanced search trees
    - Heaps and heapsort
    - Polynomial evaluation by Horner's rule
    - Binary exponentiation
3. Problem reduction = a different problem altogether
    - Lowest Common Multiple
    - Reductions to graph problem

- Pre-sorting
    - Closest pair
    - Convex hull

Space Time

Boyer Moore & Horspool

- Text of length n and Pattern P[0 ... m-1]
- Shift table called T:
    - T(X) = m-1- rightmost index of x in P[0 ... m-2]
    - T(X) = m if x is not in position P[0 ... m-2]
- Horspool
    - When there is a mismatch, shift pattern T[c] places where c is the last character currently aligned against the pattern
- Boyer-Moore's Bad Character rule
    - When there's a mismatch, calculate from the back, k, number of characters that matched
    - Shift pattern max(T(c)-k,1) where c is the mismatched character