Algorithm Design

Lec05

Dr. Mohammad Ahmad

Algorithm Design

When searching for a solution, we may be interested in two types:

- Either we are looking for the optimal solution, or,
- We are interested in a solution which is *good enough*, where good enough is defined by a set of parameters

Algorithm Design

For many of the strategies we will examine, there will be certain circumstances where the strategy can be shown to result in an optimal solution

In other cases, the strategy may not be guaranteed to do so well

Algorithm Design

Any problem may usually be solved in multiple ways

The simplest to implement and most difficult to run is *brute force*

– We consider all possible solutions, and find that solution which is optimal

Algorithm Design Techniques

- Brute Force
- Divide and Conquer
- Greedy Algorithms
- Dynamic Programming
- Backtracking

Brute Force

- Based on the problem's statement and definitions of the concepts involved.
- Examples:
  - Sequential search
  - Simple sorts: selection sort, bubble sort
  - Computing n!

Brute Force

Brute force techniques often take too much time to run

We may use brute-force techniques to show that solutions found through other algorithms are either optimal or close-to-optimal

Brute Force

With brute force, we consider all possible solutions

Most other techniques build solutions, thus, we require the following definitions

Definition:

– A *partial solution* is a solution to a problem which could possibly be extended
– A *feasible solution* is a solution which satisfies any given requirements

Algorithm Design

Thus, we would say that a brute-force search tests all feasible solutions

Most techniques will build feasible solutions from partial solutions and thereby test only a subset of all possible feasible solutions

Algorithm Design

It may be possible in some cases to have partial solutions which are acceptable (that is, feasible) solutions to the problem

In other cases, partial solutions may be unacceptable, and therefore we must continue until we reach a feasible solution

Divide and Conquer

Reduce the problem to smaller problems (by a factor of at least 2) solved recursively and then combine the solutions

Examples: Binary Search
Mergesort
Quick sort

In general, problems that can be defined recursively

Decrease and Conquer

Reduce the problem to smaller problems solved recursively and then combine the solutions

Examples of decrease-and-conquer algorithms:

Insertion sort
(recursion)
Computing Fibonacci numbers (recursion)

Greedy Algorithms

"take what you can get now" strategy

Work in phases:

In each phase the currently best decision is made.

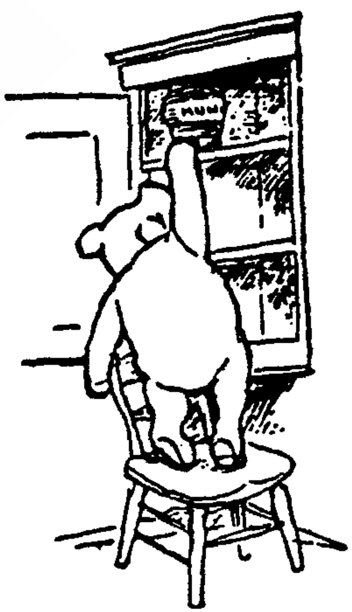
A *greedy algorithm* always makes the choice that looks best at the moment.

Greedy Solutions to Optimization Problems

Every two-year-old knows the greedy algorithm.

In order to get what you want,
just start grabbing what looks best.

Surprisingly, many important and practical optimization problems can be solved this way.



Elements of Greedy Strategy

- An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
  - NOT always produce an optimal solution
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property
  - Optimal substructure

Greedy Algorithms - Examples

- **Dijkstra's algorithm**
  (shortest path is weighted graphs)
- **Prim's algorithm, Kruskal's algorithm**

(minimal spanning tree in weighted graphs)

- **Coin exchange problem**
- **Huffman Trees**

Dynamic Programming

Bottom-Up Technique in which the smallest sub-instances are ***explicitly*** solved first and the results of these used to construct solutions to progressively larger sub-instances.

Example:

Fibonacci numbers computed by iteration.

Backtracking

Generate-and-Test methods

Based on exhaustive search in multiple choice problems

Typically used with depth-first state space search problems.

Example: Puzzles

Backtracking – State Space Search

- initial state
- goal state(s)
- a set of intermediate states
- a set of operators that transform one state into another. Each operator has preconditions and postconditions.
- a cost function – evaluates the cost of the operations (optional)
- a utility function – evaluates how close is a given state to the goal state (optional)

Conclusion

How to choose the approach?

First, by understanding the problem, and second, by knowing various problems and how they are solved using different approaches.