Advanced Algorithmic Problem Solving

Le 2 – Problem Solving Paradigms

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

Important Problem Solving Approaches

- Simulation/Ad hoc
  - Do what is stated in the problem
  - Example: Simulate a robot
- Greedy approaches
  - Find the optimal solution by extending a partial solution by making locally optimal decisions
  - Example: Minimal spanning trees, coin change in certain currencies
- Divide and conquer
  - Take a large problem and split it up in smaller parts that are solved individually
  - Example: Merge sort and Quick sort
- Dynamic programming
  - Find a recursive solution and compute it "backwards" or use memoization
  - Example: Finding the shortest path in a graph and coin change in all currencies
- Search
  - Create a search space and use a search algorithm to find a solution
  - Example: Exhaustive search (breadth or depth first search), binary search, heuristic search (A*, best first, branch and bound)

Outline

- Complete search (iterative and recursive, UVA 11656, UVA 750)
- Divide and Conquer (binary search, UVA 11935)
- Greedy search (lab 1.1, UVA 10382)
- Dynamic programming (lab 1.2, lab 1.3, UVA 147, UVA 11450, UVA 507, UVA 108)

Complete Search

- When a problem is small or (almost) all possibilities have to be tried *complete search* is a candidate approach.
- To determine the feasibility of complete search estimate the number of calculations that have to be made in the worst case.
- *Iterative complete search* uses nested loops to *generate* every possible complete solution and *filter* out the valid ones.
  - Iterating over all permutations using next_permutation
  - Iterating over all subsets using bit set technique
- *Recursive complete search* extends a partial solution with one element until a complete and valid solution is found.
  - This approach is often called *recursive backtracking*.
  - *Pruning* is used to significantly improve the efficiency by removing partial solutions that can not lead to a solution as soon as possible. In the best case only valid solutions are generated.

Example Problem: UVA 11565 and UVA 750

- UVA 11565: Iterative complete search
- UVA 750: Recursive complete search

Divide and Conquer

- Divide and conquer is very common and powerful technique which divides a problem into smaller parts, solves each part recursively and then puts together the answer from the pieces.
- Many well known algorithms are based on divide and conquer such as quick sort, merge sort and binary search.
- Binary search is a very versatile and useful technique which can be used to
  - find a particular value in a sorted range,
  - find the parameters of a (convex) function that gives a particular value,
  - find the minimum/maximum value of a function.
- Binary search can be implemented either using built in functions (lower_bound/upper_bound), iterating until the difference between the end points is small enough or iterate a constant but sufficiently large number of times.

Example Problem: UVA 11935

Greedy

- An algorithm is said to be greedy if it makes a locally optimal choice in each step towards the globally optimal solution.
- For a greedy algorithm to give a globally optimal result a problem must have two properties:
  - It has optimal sub-structures, i.e. an optimal solution contains the optimal solutions to sub problems.
  - It has the greedy choice property, i.e. if we extend a partial solution by making a locally optimal choice we will get the optimal complete solution without reconsidering previous choices.
- Classical examples: Coin change in some currencies, interval coverage and load balancing.
- Greedy algorithms can be very useful as heuristics for example in branch-and-bound search algorithms.
- In combinatorics matroids and the generalization greedoids characterize classes of problems with greedy solutions.

Example problem: UVA 10382

Dynamic Programming

- Dynamic Programming is a problem solving approach which computes the answer for every possible *state* exactly once.
- For DP to be suitable a problem must have two properties:
  - It has optimal sub-structures, i.e. an optimal solution contains the optimal solutions to sub problems.
  - Overlapping sub-problems, i.e. the same subproblem occurs many times.
- Top-down (memoization) vs Bottom-up
  - Top-down: no need to consider the order of computations, only compute states actually used, natural transition from complete search,
  - Bottom-up: no recursion, computes every state, table size can be reduced if only the previous row of states is used then only two rows are required.
- Displaying the optimal solution
  - Store the previous state for each solution
  - Use the DP table and the optimal sub-structures property to compute the path.

Example problem: UVA 11450

Example problem: UVA 507 and UVA 108

Classical DP Problems

- Max 1D sum
- Max 2D sum
- Longest increasing subsequence (LIS)
  - Longest decreasing subsequence (LDS)
- 0-1 Knapsack (subset sum)
- Coin Change (general version)
- Travelling Salesman Problem (TSP)

	1D RSQ	2D RSQ	LIS	Knapsack	CoinChange	TSP
State	(i)	(i,j)	(i)	(id,remW)	(v)	(pos,mask)
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all j<i	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$