An Introduction to Algorithmic Problem-solving Techniques

Question: Given a computational task, how do we devise algorithms to solve it?

One answer: Algorithmic techniques which may or may not provide solutions: a range of tools for constructing algorithms.

We look briefly at three techniques, introducing you to the ideas. Laboratory exercises will develop these techniques:

- Divide-and-Conquer
- Greedy Strategies
- Dynamic Programming

This is Part III (Chapters 10, 11 and 12) of the course textbook.

Problem: Consider finding both the maximum and minimum of a sequence of integers.

The obvious method is to find the maximum by iterating through the sequence and then doing the same to find the minimum.

Question: What is the time complexity of this? How many integer comparisons do we make for a sequence of length N?

Answer: For the maximum it is $N - 1$ comparisons, and then for the minimum it is $N - 2$ comparisons, so the total is

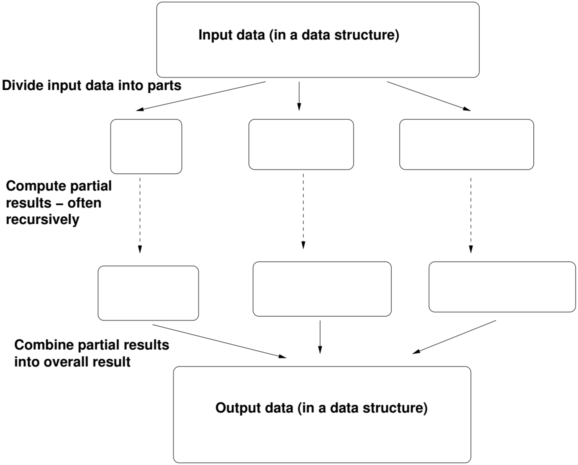
$$2N - 3$$

comparisons.

Question: Can we do better?

Divide-and-Conquer - The Idea

The scheme is: Divide the input into parts, solve the parts (often recursively), then combine the solutions to give the final result.



Divide-and-Conquer - Example

A divide-and-conquer solution for max-and-min problem

Divide the sequence s into s_1 and s_2 (arbitrarily – any elements in either subsequence, but approximately equal sizes for efficiency).

Now recursively calculate the maximum and minimum of subsequence s_1 (call them max_1 and min_1) and of subsequence s_2 (max_2 and min_2).

Can we calculate the maximum and minimum of the whole sequence s?

Divide-and-Conquer - Example

A divide-and-conquer solution for max-and-min problem

Divide the sequence s into s_1 and s_2 (arbitrarily – any elements in either subsequence, but approximately equal sizes for efficiency).

Now recursively calculate the maximum and minimum of subsequence s_1 (call them max_1 and min_1) and of subsequence s_2 (max_2 and min_2).

Can we calculate the maximum and minimum of the whole sequence s?

Yes: the maximum is $max(max_1, max_2)$ and the minimum is $min(min_1, min_2)$.

Divide-and-Conquer - Example (continued)

This gives a recursive algorithm:

```
maxmin(s) =
    if s = [x] then return (x,x);
    if s = [x1,x2]
     then
        if x1>x2 then return (x1,x2) else return (x2,x1);
     else
      (s1,s2) = divide(s);
      (max1,min1) = maxmin(s1);
      (max2,min2) = maxmin(s2);
      return (max(max1,max2),min(min1,min2))
```

Divide-and-Conquer - Example (continued)

Question: Is this any better? How many integer comparisons are required for a sequence of length N? This is hard to say!

Divide-and-Conquer - Example (continued)

Question: Is this any better? How many integer comparisons are required for a sequence of length N? This is hard to say!

Let C_N be the number of integer comparisons needed on a sequence of length N. Then

$$C_N = 2 \times C_{\frac{N}{2}} + 2$$

and $C_1 = 0$ and $C_2 = 1$.
Why? – Count the operations at each stage of the algorithm.

The solution of this is

$$C_N = \frac{3N}{2} - 2.$$

This is much smaller than the previous method, which required $2N - 3$. Why are fewer comparisons needed? Notice that both methods are linear, $O(N)$.

Divide-and-Conquer - Applications

Divide-and-conquer is of very wide application. A few examples:

- Efficient sorting algorithms - both Mergesort and Quicksort have average time complexity of $O(N \times \log(N))$, whereas most simple general sorting algorithms are slower, $O(N^2)$.
- Fast integer multiplication: Integer multiplication by long multiplication is $O(N^2)$, but there are fast $O(n \times \log(N))$ divide-and-conquer algorithms.
- Fast matrix multiplication: Standard matrix multiplication is $O(N^3)$, divide-and-conquer algorithms produce algorithms $O(N^{2.808\ldots})$ (and even down to $O(N^{2.376\ldots})$).
- Nearest neighbour problems: Given a set of points in 2D or 3D (or N dimensions), find two nearest points. Divide-and-conquer algorithm is $O(N \times \log(N))$.

Greedy Methods - An Example

Problem: Suppose we have a set of coins of various denominations (values) and we wish to pay for an item of cost V with a minimum number of coins.

How do we select the coins to do this?

Suppose (as in the UK) we have coins of values 1, 5, 10 and 20 pence, and we wish to pay for an item costing 37 pence.

We need to choose a minimum number of coins to do this. How?

Answer: Choose the biggest value coins that we can at each stage: choose a 20, then a 10, then a 5, then a 1, then a 1, and then we are done! 5 coins are needed and this is a minimum.

This is called a greedy strategy.

Greedy Methods - An Example (continued)

Question: Does a greedy strategy always work for this problem?

For these coin values, it always works. Why? Answer is not obvious!

Consider coin values 1, 10 and 6 and we have item costing 12.

Then the greedy strategy fails. We choose a 10 and then two 1s, to give 3 coins. But we could have chosen 2 coins of value 6.

Greedy Methods - Optimisation Problems

An Optimisation Problem requires us not simply to solve the problem, but to produce a 'best' solution.

'Best' is in terms of some evaluation of the quality of the solution. It may mean the largest or smallest, the closest or furthest, the shortest or longest, etc. In general, we talk of maximum and minimum solutions.

A greedy strategy attempts to find a maximum (or minimum) solution, by maximising (or minimising) the value of the choice at intermediate stages.

Applications

- For some optimisation problems a greedy strategy is always successful.
- For some optimisation problems, a greedy strategy solves some instances but not others (as in the coins example above).
- For some optimisation problems, a greedy strategy may not give optimal solutions but may lead to good approximations to optimal solutions.
- For some optimisation problems, a greedy strategy is not applicable – no useful solutions are produced.

Some standard problems with greedy solutions

- Many path-finding algorithms - eg shortest paths using Dijkstra's algorithm.
- Some job scheduling problems admit greedy solutions, others don't.
- Spanning trees of a graph: Given an edge-weighted graph, choose a subset of the edges which form a tree and which include all nodes . We seek such a tree with minimum combined edge-weights.
- Knapsack problems: The fractional knapsack problem admits a greedy solution, but for the 0/1 problem, greedy solutions are not necessarily optimal.

Dynamic Programming - A Simple Example

Question How do we solve the general case of the coin problem?

Greedy solutions fail in general.

Dynamic Programming - A Simple Example

Question How do we solve the general case of the coin problem?

Greedy solutions fail in general.

Dynamic programming always produces an optimal solution to this problem.

Suppose we have coin types $1, \ldots, N$ and the value of coin type i is v_i.

Suppose that we have enough coins of each type (if we have a limited number of coins of each type, we can modify the idea below).

How do we make a sum with a minimum number of coins?

Dynamic Programming - A Simple Example (continued)

Obvious property:

Let $c(i, s)$ be the minimum number of coins from types 1 through to i required to make sum s. Consider what happens if we add another coin type $i + 1$:

$$
\begin{array}{rll}
c(i+1, s) = min(& c(i, s) & \\
 & c(i, s - v_{i+1}) + 1 & \\
 & \vdots & \\
 & c(i, s - k \times v_{i+1}) + k \;\;) & \text{where } (k+1)v_{i+1} > s \\
c(i, s) = \infty & \text{if value } s \text{ cannot be made with coins } 1 \dots i. &
\end{array}
$$

This says: if we have solved the problem for coins of types $1 \dots i$ and now we consider coins of type $i + 1$, then an optimal solution may be to use no coins of type $i + 1$ or one such coin combined with an optimal solution of a smaller problem using coin types $1 \dots i$, or two coins of type $i + 1$...

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	3
coin types 1,2,3,4	1	2	3	4	1	1	2	3	1	2	2

Answer: Using all 4 coin types, 2 coins is the minimum number needed to make a sum of 11.

Dynamic Programming - Notes

Dynamic programming is a bottom-up method – we solve all smaller problems first then combine them to solve the given problem.

Question: How efficient is this dynamic programming solution? What is its time complexity?

The main factor is the size of the table of subproblem results. Thus for N coin types, and a value required of V, the table size is $N \times V$ (notice that this is in terms if N and V, where it is natural to ask for the dependence on V only).

Notice that some subproblem results are not required for the final solution - but this is not easy to use as a reduction strategy: it is difficult to predict what might be needed.

Dynamic Programming - Applications

A few examples of optimisation problems with dynamic programming solutions

- Some path-finding algorithms use dynamic programming, for example Floyd's algorithm for the all-nodes shortest path problem.
- Some text similarity tests: For example, longest common subsequence.
- Knapsack problems: The $0/1$ Knapsack problem can be solved using dynamic programming.
- Constructing optimal search trees.
- Some travelling salesperson problems have dynamic programming solutions.