

Topic 19: Computational Thinking and Problem-Solving

Dynamic programming, graph traversal and the limits of what can be computed.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 19.1 Algorithms (recursion, Big O, graphs), 19.2 Recursion. Try the challenge questions on the last pages before you read the answers.

Dynamic programming: don't solve the same problem twice

Greedy choices can be wrong. With coins {1, 3, 4} and a target of 6, grabbing the largest coin first gives $4+1+1 = 3$ coins, yet $3+3 = 2$ coins is better. **Dynamic programming** solves it by building answers for small amounts up to the target: $\text{best}(n) = 1 + \text{the smallest best}(n - \text{coin})$ over all coins that fit.

Amount	0	1	2	3	4	5	6
Fewest coins	0	1	2	1	1	2	2

```

best[0] <- 0
FOR n <- 1 TO Target
  best[n] <- n // upper bound: n one-coin steps
  FOR EACH Coin IN Coins
    IF Coin <= n AND best[n - Coin] + 1 < best[n] THEN
      best[n] <- best[n - Coin] + 1
    ENDIF
  NEXT Coin
NEXT n

```

This **bottom-up** (tabulation) method uses the same insight as memoisation (store sub-results) but without recursion. It needs *optimal substructure* (the best answer contains best answers to sub-problems) and *overlapping sub-problems*.

Walking a graph: breadth-first vs depth-first

Graphs model maps, networks and dependencies. Two ways to visit every node: **breadth-first search (BFS)** uses a *queue* and explores all neighbours before going deeper, so it finds the route with the fewest edges; **depth-first search (DFS)** uses a *stack* (or recursion) and follows one path as far as it goes before backtracking. Both take $O(V + E)$ time.

```

Graph:  A -> B, C      B -> D      C -> D, E      D -> F      E -> F

BFS from A:  A B C D E F      (level by level)
DFS from A:  A B D F C E      (deep first, then backtrack)

```

DFS suits cycle detection, maze solving and ordering tasks by dependency; BFS suits shortest-hops problems, e.g. social-network "degrees of separation".

The limits of computation

Hard but solvable. The travelling salesman problem asks for the shortest tour through n cities. There are $(n-1)!/2$ tours: 181 440 for 10 cities but about 6×10^{16} for 20. No known algorithm solves every case quickly; this class of problem is bound up with the famous open question **P vs NP**. In practice we use heuristics that find good, not guaranteed-best, tours.

Impossible. The **halting problem** asks for a program $\text{Halts}(P, I)$ that decides whether program P ever finishes on input I . Turing showed none can exist. Sketch: suppose Halts exists and write $\text{Paradox}(P)$: *if $\text{Halts}(P, P)$ says "finishes", loop forever; otherwise stop*. Now run Paradox on itself. If Halts says it finishes, it loops; if Halts says it loops, it stops. Contradiction. Some questions about programs can never be answered by any program.

Challenge questions

1. Coins $\{1, 5, 6\}$, target 10. What does the greedy method give, and what is the true minimum?
2. Why does BFS find the route with the fewest edges in an unweighted graph, while DFS may not?
3. How many distinct tours are there for 12 cities? At 10^9 tours checked per second, estimate the time for 25 cities.

Worked answers

Only look once you have had a proper go at each question.

1. Greedy: $6 + 1 + 1 + 1 + 1 = 5$ **coins**. Optimal: $5 + 5 = 2$ **coins**. Dynamic programming finds $\text{best}[10] = 1 + \text{best}[5] = 2$.
2. BFS visits nodes in order of distance from the start (all nodes one edge away, then two, ...), so the first time it reaches the target it has used the fewest edges. DFS commits to one long path first and may reach the target by a much longer route.
3. 12 cities: $11!/2 = 19\,958\,400$ tours (checked in about 0.02 s).
25 cities: $24!/2 \approx 3.1 \times 10^{23}$ tours; $\div 10^9 \approx 3.1 \times 10^{14}$ seconds $\approx$ **10 million years**. Brute force is hopeless.