A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

Traveling Salesman

Specifications. State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, distinguish between legal solutions and optimal ones.

> Given a list of cities and the distances between each pair of cities, find the shortest route that visits each city exactly once and returns to the origin city.
>
> Input: n cities, distances $d_{i,j}$ between each pair i, j of cities
>
> Output: route (ordering of cities)
>
> Legal solution: each city is included exactly once
>
> Optimization goal: shortest total distance

Examples. If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

> The size of the problem is n, the number of cities.
>
> The smallest problem is $n = 0$.

Targets. What are the time and space requirements for your solution?

> This is an ordering problem — there are $O(n!)$ possible permutations of the cities.

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

> Since subsequent choices of cities depend only on what hasn't yet been visited, this seems like a candidate for dynamic programming being able to improve the running time.

Approaches. Identify the particular flavor, if applicable, as well as the two patterns (process input and produce output) and what they look like for this problem.

> This is an ordering problem.
>
> Process input — for each city, determine where it goes in the route
>
> Produce output — find the next city in the route

Both approaches have a similar branching factor, and produce output has the advantage of being able to append each new thing to the route so far.

Generalize / define subproblems. Friends get smaller versions of the original problem. Define the task, input, and output for the subproblems.

Partial solution. Identify what constitutes a solution-so-far. This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

> The whole solution is an ordering of all n cities, so a partial solution is an ordering of the first k cities.

Alternatives. Identify the choice being made and what the (legal) alternatives are for that choice. (Alternatives that result in an illegal solution aren't considered farther.)

> The choice is the next city to visit. All not-yet-visited cities are legal alternatives.

Subproblem. The "rest of the problem". Define the generalized problem, its input, and its output. The input can include the partial solution so far, in which case the output would be a complete solution, or the input can only denote the subproblem, in which case the output is only the solution for the subproblem.

> Original problem: Given a list of cities and the distances between each pair of cities, find the shortest route that visits each city exactly once and returns to the origin city.
>
> Subproblem / generalized problem: Given a list of cities, the distances between each pair of cities, and the route so far, find the shortest route that visits the rest of the cities exactly once and returns to the origin city.
>
> Input: set C of $n-k$ cities remaining, distances $d_{i,j}$ between each pair of cities, route so far R
>
> Output: route (ordering of cities) and the total distance of that route
>
> Legal solution: each remaining city is included exactly once
>
> Optimization goal: shortest total distance

Base case(s). A base case occurs when there are no more choices to make.

> There are no more choices to make when we've visited all of the cities — $n-k=0$. Return an empty ordering and the distance between the last city visited ($R[n-1]$) and the origin city ($R[0]$).

Main case. This takes the form of: for each legal alternative for the current decision, a friend solves the remainder of the problem given the solution so far plus that choice. We return one of those solutions, the best of those solutions, or all of the those solutions depending on the structural variation.

> `for each city` c `in` C
>
> $(\texttt{route}_c, \texttt{dist}_c)$ `= tsp(`$C-\{c\}$`,`R `with` c `appended)`
>
> `return` $c+\texttt{route}_c$ `and` $d_{R[k-1],c}+\texttt{dist}_c$ `for the city` c `for which` $\texttt{dist}_c+d_{R[k-1],c}$ `is maximized`

Top level. The top level puts the context around the recursion.

> **Initial subproblem.** C is the full list of cities except for c_o, R is c_0.
>
> **Setup.** Choose a city c_0 to be the origin city.
>
> **Wrapup.** Return just the ordering found, not the length.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for special cases as needed.

The initial subproblem assumes $n \geq 1$. For $n = 0$ the solution is an empty sequence with a total distance of 0.

Termination. Show that the recursion ends. For making progress, explain why each subproblem is smaller. For reaching the end, show that a base case is always reached.

Making progress. The number of cities left is $n-k$, and this decreases by 1 with each recursive call.

Reaching the end. The base case is when $n - k = 0$, and $n - k$ decreases by 1 each time.

Establish the base case(s). Explain why the solution is correct for each base case.

The city removed from the list remaining is added to the route, so when $n - k = 0$, the length of R is n — a complete solution. Only legal alternatives were considered, so a complete solution is a legal solution.

Show the main case. Explain why all possible alternatives for the next choice are covered and that the right partial solution is passed to each friend. Then, assuming that the friends return the correct results for their subproblem, explain why the correct answer is produced from those results.

The remaining cities are exactly those not yet in the route, and all are considered as the next city in the route.

c is removed from the remaining cities and appended to the route so far for each friend. The solution returned prepends c to the friend's route and adds the distance from the last city in R to c to the total distance of the friend's route.

Final answer. Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

Since the route is a cycle, any city c_0 can be the origin. Thus `tsp` with c_0 as the route so far and all of the cities except c_o left to visit covers the whole problem.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

Memoization. Identify how to parameterize subproblem state for efficient lookup, typically in an array.

The input for the subproblems is the set C of $n-k$ cities remaining, the distances $d_{i,j}$ between each pair of cities, and the route so far R.

The $d_{i,j}$ is global information common to all subproblems — it is not part of the memoization.

Of the route so far, only the last-visited city is needed (for the distance between it and the next city). (The origin is also global information common to all subproblems.)

The set C of remaining cities can't be represented by a single index as in longest increasing subsequence — the next alternatives are all of remaining cities, so every subset is needed. A set can be represented with an integer — consider the binary representation of the integer, with each city corresponding to a particular bit. The bitwise operators allow for efficient add/remove operations in the set, but a set of n things still requires n bits = 2^n different values.

Let `D[C][r]` be the total distance of the shortest route continuing from r and involving all of the vertices in the set C (represented as an integer).

Order of computation. Loop(s) can be used to fill in the array; identify whether the loop(s) go from small large index values or vice versa.

> The base case is $n - k = 0$ so `C` needs to be filled in from small to large. `D[C][]` only depends on smaller values so the order for r isn't important.

Dynamic programming. Combine the memoization and order of computation with the base and main cases and the top level.

> `tsp(S)` — Given a list of cities, the distances between each pair of cities, and the route so far, find the shortest route that visits the rest of the cities exactly once and returns to the origin city.
>
> Input: n cities, distances $d_{i,j}$ between each pair i, j of cities
>
> Output: route (ordering of cities)
>
> Legal solution: each city is included exactly once
>
> Optimization goal: shortest total distance

```
// initialize the base cases - k=n
for C = 0 to 2^n do
  D[C][c_0] = 0

// fill in the rest of the array
for C = n-1 downto 0 do
  for r = 0 to n-1 do
    D[C][r] = maax(D[C-c][c])

// solution
return the items for D[C][c_0] where C corresponds to the set of all of the cities
  other than c_0
```

Time and space. Assess the running time and space requirements of the algorithm given the implementation identified.

> The array has $n \times 2^n$ slots and computing each value is $O(n)$ ($n - k$ alternatives to consider), so the total running time is $O(n^2 2^n)$.