NOI.PH Training: DP 2

Dynamic Programming on Special Structures

Vernon Gutierrez, Kevin Atienza

Contents

1 DP with Bitmasking

1.1 The Traveling Salesman Problem

We begin with a motivating problem.

Problem 1.1 (Travelling Salesman Problem)**.** A salesman wants to travel across a given set of n different cities, presumably in order to sell stuff. For all pairs of cities u and v, the cost of traveling from city u to city v is given. This cost may be different for different pairs of cities. The salesman wants to visit every city at least once. He can start from any city and end at any city. What is the minimum total cost to do so?

This is a classic problem in computer science. There is a simple $\mathcal{O}(n \cdot n!)$ algorithm for solving it.

Exercise 1.1. Describe an $\mathcal{O}(n \cdot n!)$ algorithm for solving this problem.

No one has found a way to solve this problem in polynomial time. Most computer scientists believe it can't be done, but no one can prove it either. However, it's not the end of the story for this problem. There is a way to solve it in $\mathcal{O}(n^2 \cdot 2^n)$, which while still not practical for large values of n, is at least a nice improvement over $\mathcal{O}(n \cdot n!)$.

The solution is to use dynamic programming. Let $opt(u, S)$ be the optimal way to visit all the cities, starting from u, assuming none of the cities in the set S have been visited yet. Since we don't know where best to go next from u, we'll just try all possibilities, recursively solve the remaining problem, and take the best one. There is no point in going back to a city we've already visited. We might as well try directly going to a city which we haven't visited yet: choose a city v that is in S. On our way from u to v, passing through cities we have already visited is alright, but we don't want to incorporate these "micro-decisions" into our DP state. In fact, we'll just assume we already computed the shortest paths between any two cities. We definitely always want to take the shortest path from our current starting city to our target destination. We'll denote the total cost of the shortest path from u to v as $w(u, v)$. Every time we visit a new city, we remove it from S. If S is empty then we've already visited all the cities. In that case, the cost is 0. In summary:

$$opt(u, S) = \begin{cases} 0 & S \text{ is empty} \\ \min\limits_{v \in S} w(u, v) + opt(v, S - \{v\}) & \text{otherwise} \end{cases}$$

Since we can start at any city, the final answer that we want is

$$\min_{v \in V} opt(v, V - \{v\})$$

where V is the set of all cities.

Alternatively, if we label the cities with integers from 1 to n, we can introduce a "dummy" city 0 and define the cost of directly going from city 0 to city v to be 0 for all v. The final answer is then just $opt(0, V - \{0\})$. We'll use this to simplify the implementation.

We can directly implement the recurrence above as follows.

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_N = 20;
int w[MAX_N+1][MAX_N+1];
int opt(int u, set<int> S) {
    if(S.empty()) {
        return 0;
    } else {
        int ans = numeric_limits<int>::max();
        for(int v : S) {
            set<int> new_s(S.begin(), S.end());
            new_s.erase(v);
            ans = min(ans, w[u][v] + opt(v, new_s));
        }
        return ans;
    }
}
int main() {
    int n;
    // some code here for reading the input
    // some code here for computing shortest paths
    vector<int> v(n);
    iota(v.begin(), v.end(), 1);
    cout << opt(0, set<int>(v.begin(), v.end())) << endl;
    return 0;
}
```

Exercise 1.2. Calculate tight bounds for the running time of this solution.

Exercise 1.3. Give a counterexample of the traveling salesman problem for which the following recurrence doesn't work:

$$opt(u, S) = \begin{cases} 0 & S \text{ is empty} \\ \min\limits_{v \in S} c(u, v) + opt(v, S - \{v\}) & \text{otherwise} \end{cases}$$

where $c(u, v)$ is the cost of traveling *directly* from u to v.

Exercise 1.4. In our solution above, we said:

> On our way from u to v, passing through cities we have already visited is alright, but we don't want to incorporate these "micro-decisions" into our DP state.

Why not? Can't we just say that starting from u, we try all possible next destinations v, *including those not in* S, and instead of taking the shortest path from u to v, we just directly travel from u to v? Wouldn't the recursive formulation automatically take the shortest paths even to cities that have already been visited? In other words, what is wrong with the following recurrence?

$$opt(u,S) = \begin{cases} 0 & S \text{ is empty} \\ \min\limits_{v \in V} c(u,v) + opt(v, S - \{v\}) & \text{otherwise} \end{cases}$$

where V is the set of all cities.

Exercise 1.5. In our recurrence above, it's possible that the shortest path from u to v passes through cities that we haven't visited yet. However, we don't bother removing them from S when we go from u to v. Why is this fine?

Are there overlapping subproblems in our current solution? Let's say we first try visiting cities A, B, C, D in some branch of the recursion tree, in that order. We end up in the state $(D, S - \{A, B, C, D\})$. Well, any branch of the tree that first visits some permutation of A, B, and C and then visits D will end up in the same state. Clearly, there are many overlapping subproblems, and the smaller S is, the more paths there are from the root of the recursion tree leading to it.

Indeed, the way the recurrence is written won't require a computer to "remember" the exact order of cities visited in order to finish a branch of the computation. "Forgetting" about this order enables avoiding the $\mathcal{O}(n!)$ factor of running time present in purely brute-force solutions.

Exercise 1.6. Given a fixed set of cities, how many distinct input values can the recurrence above take, as a function of n, the number of cities?

It thus makes sense to memoize. Since the parameters to our function are not plain integers, we need to use a `map` for our memo.

Exercise 1.7. Before reading the next page, try memoizing the function we wrote above by yourself first.

```cpp
map<pair<int, set<int>>, int> memo;
int opt(int u, set<int> S) {
    if(not memo.count({u, S})) {
        int ans;
        if(S.empty()) {
            ans = 0;
        } else {
            ans = numeric_limits<int>::max();
            for(int v : S) {
                set<int> new_s(S.begin(), S.end());
                new_s.erase(v);
                ans = min(ans, w[u][v] + opt(v, new_s));
            }
        }
        memo[{u, S}] = ans;
    }
    return memo[{u, S}];
}
```

This solution indeed works in $\mathcal{O}(n^c \cdot 2^n)$, where c is some small constant.

Exercise 1.8. Calculate tight bounds for the running time of this solution.

However, this is not yet the best we can do.

Notice that deleting items from a set yields an extra logarithmic factor overhead in the running time. Since our program is expected to work only on small input sizes, this turns out to be not too significant.

There are two more serious issues.

Every time we transition from one state to another, we need to copy a `set`. This costs an extra linear factor of running time. We can avoid this by copying by using pass-by-reference and "undoing" moves at every branch, similar to how we do it for backtracking. But this turns out to not matter. Whenever we lookup the memo, we need to compare the stored values against a $\Theta(n)$-sized key. Hence we still incur an extra linear factor simply because we represent sets using a `set` object.[1]

Another serious issue is that, since we're using a `map` for the memo, we spend an extra logarithmic factor to access it. It's not logarithmic in n, but logarithmic in the *total number of subproblems*, which is roughly 2^n. In other words, we actually spend an extra $\mathcal{O}(\log 2^n) = \mathcal{O}(n)$ factor for using a `map`. This is much more significant than the logarithmic factor due to using the `set`.

To make the implementation faster, we need to find a more *compact* way to represent sets, or more accurately speaking, *subsets* of a fixed set of n elements.

If you remember the trick used to iteratively complete-search through all subsets of a given set using just integers and bitwise operations, then that's exactly what we're going to do. Recall that every subset S of a fixed set of integers from 0 to $n-1$ can be uniquely mapped into an integer using the following rule: represent S using the integer $b_0 \cdot 2^0 + b_1 \cdot 2^1 + \cdots + b_{n-1} \cdot 2^{n-1}$, where b_i is 1 if i is in S, 0 otherwise. We call this integer the **bitmask**.

Using some bitwise black magic$^{\text{TM}}$, we can write the set operations that we want in terms of bitwise operations:

[1]More generally, the lesson here is that passing "heavy" data structures around is inefficient. It's true that this extra inefficiency is dwarfed by the $\mathcal{O}(2^n)$ factor due to the sheer number of states we have to compute, but we don't want to make an already slow algorithm even slower.

```cpp
constexpr int num_subsets(int n) { return 1 << n; }
bool empty(int S) { return S == 0; }
int erase(int S, int v) { return S & ~(1 << v); }
int all(int n) { return (1 << n) - 1; }
```

We can now rewrite our `opt` function to take only integers as input, and we can memoize using plain arrays:

```cpp
int n;
const int MAX_N = 20;
int w[MAX_N+1][MAX_N+1];
int memo[MAX_N+1][num_subsets(MAX_N)];
int opt(int u, int S) {
    int& ans = memo[u][S];
    if(ans == -1) {
        if(empty(S)) {
            ans = 0;
        } else {
            ans = numeric_limits<int>::max();
            for(int v = 1; v <= n; v++) {
                ans = min(ans, w[u][v] + opt(v, erase(S, v)));
            }
        }
    }
    return ans;
}
int main() {
    // some code here for reading the input
    // some code here for computing shortest paths
    cout << opt(0, erase(all(n + 1), 0)) << endl;
    return 0;
}
```

When you get used to working with bitmasks, it's not necessary to write the helper functions. But they're helpful for ease of understanding and they're less error-prone. Most likely, the code you will read and write in competitions will look more like this:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 20;
int n;
int w[MAX_N+1][MAX_N+1];
int memo[MAX_N+1][1 << MAX_N];
int opt(int u, int S) {
    int& ans = memo[u][S];
    if(ans == -1) {
        if(S == 0) {
            ans = 0;
        } else {
            ans = numeric_limits<int>::max();
            for(int v = 1; v <= n; v++) {
                ans = min(ans, w[u][v] + opt(v, S & ~(1 << v)));
            }
        }
    }
    return ans;
}
int main() {
    // some code here for reading the input
    // some code here for computing shortest paths
    cout << opt(0, (1 << (n + 1)) - 2) << endl;
    return 0;
}
```

I used to wonder how people could write such strange code. Now I know! And so do you!

Exercise 1.9. Assume there can be at most 20 cities in the input. The code above has a bug. Find it.

Exercise 1.10. What should we do if we are dealing not with the set of integers from 0 to $n-1$, but with just some arbitrary set of n items?

Exercise 1.11. A **Hamiltonian path** in a graph is a path that contains each node of the graph exactly once. Design an algorithm that determines whether there is a Hamiltonian path in a graph in $\mathcal{O}(n \cdot n!)$ time. Then, design an algorithm that solves the same problem in $\mathcal{O}(n^2 \cdot 2^n)$ time.

DP with bitmasking is generally a helpful technique for reducing $\mathcal{O}(n!)$ factors to $\mathcal{O}(2^n)$. Whenever the problem you face has an obvious "complete-search through all permutations" solution, and the constraints are on the small side – slightly too large for $\mathcal{O}(n^c \cdot n!)$ solutions to pass but small enough for $\mathcal{O}(n^c \cdot 2^n)$ solutions to pass, then DP with bitmasking might be the right approach.

If the $\mathcal{O}(n^c \cdot n!)$ algorithm is not so obvious, start with coming up with it anyway, even if it's too slow. After coming up with an $\mathcal{O}(n^c \cdot n!)$ algorithm, it may become easier to see the DP with bitmasking solution that runs in $\mathcal{O}(n^c \cdot 2^n)$.

Exercise 1.12. Minimum Feedback Arc Set. Design an algorithm to solve the following problem: given a directed graph, find the minimum number of edges to remove to make it acyclic.

2 DP and Acyclic Graphs

Let's now step back a bit and get a more birds-eye view of DP and why it works. You've pretty much seen that there is a relationship between DP and recursion. We might even say that DP is approximately the "art of finding a good recurrence."

But not all recurrences are amenable to DP! Recall Exercise 1.4. I assume you've already done that exercise, but if you haven't (try it first!), then one way we can figure out what goes wrong is to just implement it. Implementing it using memoization and then running it on a small-ish graph quickly reveals the issue: the program crashes with a stack overflow error, suggesting that it goes into infinite recursion!

In fact, the main issue with the recurrence in Exercise 1.4 is that the transitions are *cyclic*. For example, suppose there is a path from A to B and vice versa. Then the states $(A, V - \{A, B\})$ and $(B, V - \{A, B\})$ are reachable from each other. But this means our DP program will go back and forth between these states, which is why it goes into infinite recursion. This clearly happens in general whenever there's a cycle among the possible states.

In fact, the recurrence stated in Exercise 1.4 is correct. It's just that it cannot be used as the recurrence of a DP or memoization solution because there are cycles in the states. It's analogous to saying that the Fibonacci numbers satisfy the following recurrence:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 2014, \\ F_{n+1} - F_{n-1} & \text{if } n \le 2014. \end{cases}$$

This is clearly true, but useless as a DP recurrence.

The above discussion illustrates an important requirement for DP: **the states must be acyclic**.

In retrospect, we can now see why many of the problems we've solved with DP are even solvable with DP at all: If you take a look at the recurrences, there's usually a variable (or expression) that always decreases in the recurrence, so there can be no cycles among the states. For example, in the recurrence we obtained for the coin change problem, namely

$$C(i) = \min(C(i-1) + 1, C(i-10) + 1, C(i-25) + 1),$$

notice that $i - 1$, $i - 10$ and $i - 25$ are all less than i, so there is no way to return to the same state. (The program can still go into infinite recursion if there's no base case, so base cases are still needed.)

Exercise 2.1. Consider the following problem:

Problem Statement: *There is a grid or $r \times c$ cells. We denote by (i, j) the cell at row i and column j, with $1 \le i \le r$ and $1 \le j \le c$. Suppose you start at cell $(1, 1)$ and wish to go to cell (r, c). The cost of stepping on cell (i, j) is $A_{i,j}$. Starting at cell (i, j), there are only four allowed moves:*

- $(i, j) \to (i - 1, j + 2)$,
- $(i, j) \to (i + 1, j + 2)$,
- $(i, j) \to (i + 2, j + 1)$,
- $(i, j) \to (i + 2, j - 1)$.

In other words, the allowed moves are four of the eight possible knight moves. It is not allowed to go off the grid. What is the minimum cost to go from $(1, 1)$ to (r, c)?

Find a recurrence for this problem, and then explain why it's amenable to DP/memoization, i.e., there are no cycles in the states.

Exercise 2.2. Explain why there are no cycles among the states visited by our bitmask DP solution to the Travelling Salesman Problem.

2.1 DP on DAGs: The Longest Path Problem

Let's look at the previous discussion from another viewpoint. We've seen that DP requires the states to form an acyclic structure. But thinking about it from the opposite viewpoint, this suggests that problems on things that naturally have an acyclic structure (e.g., trees, or DAGs) are particularly approachable with DP!

To illustrate this, let's look at the following problem, which can be contrasted with the Travelling Salesman Problem:

Problem 2.1. A not-so-bright salesman wants to travel across a given set of n different cities, presumably in order to sell stuff. For all pairs of cities u and v, the cost of traveling from city u to city v is given. This cost may be different for different pairs of cities. The salesman only wants to visit each city at *most* once. He can start from any city and end at any city. What is the *maximum* total cost to do so?

We want to call this the Travelling Scammer Problem. Unfortunately, it already has an official name! It's called the **Longest Path Problem**, which is actually a better name anyway, so let's use that.

Just like the Travelling Salesman Problem, no one knows any polynomial-time solution to the Longest Path Problem, but no one has proven that there aren't any either.[2] Also, there's an obvious $O(n^c \cdot n!)$ brute-force solution, which can be improved to $O(n^c \cdot 2^n)$ using DP with bitmasking. Our state will be (u, S), where u is the city that the salesman is currently in, and S is the set of cities that he has already visited.[3] Our recurrence will be:

$$opt(u, S) = \max\left(0, \max_{v \in \text{adj}(u) - S} c(u, v) + opt(v, S \cup \{v\})\right) \tag{1}$$

where $\text{adj}(u)$ is the set of cities *directly* reachable from u, and $c(u, v)$ is the cost of traveling *directly* from u to v. The 0 in this formula represents the fact that we can stop at any time. The answer then is

$$\max_{v \in V} opt(v, \{v\}).$$

Exercise 2.3. Show that the previous recurrence doesn't run into infinite recursion. *Hint:* You need to show that there are no cycles in the states, and that there are base cases.

[2]The Travelling Salesman Problem and the Longest Path Problem are both tied to the famous problem called **P vs NP**. One of the fundamental results in computer science (in particular, complexity theory) is that P = NP iff the Travelling Salesman Problem can be solved in polynomial time iff the Longest Path Problem can be solved in polynomial time.

[3]We can equally use the state (u, S) where S is the set of cities that he has yet to visit. The recurrence will be different, but highly similar.

Can we do better? Unfortunately, we seem to be stuck with at least exponentially many states because we need to consider the possibilities of which sets of cities we've already visited, and there can be exponentially many of them; in fact, up to 2^n in the worst case if there is a road from any city to any other city. And we seem to *need* to remember which sets we've already visited so we know we can't visit them anymore.

However, there's a special case of the Longest Path Problem that can be solved easily, which is the Longest Path Problem on a Directed Acyclic Graph (DAG). Recall that the defining property of a DAG is that *there are no cycles among the nodes.* Notice that this is precisely the requirement for DP to work! In our case, this means we don't actually need to keep track of the set of already-visited cities, because we already know that *there's no way to revisit any city* at all! This means that all we need to remember is which city we're currently on.

This gives us the following simplified recurrence:

$$opt(u) = \max\left(0, \max_{v \in \mathrm{adj}(u)} c(u,v) + opt(v)\right)$$

This is basically the same recurrence as before; we simply dropped S from our state. The answer then is

$$\max_{v \in V} opt(v).$$

It should be readily apparent why this works—we've basically taken advantage of the acyclicity of a DAG. Also, it's clear that the running time is *linear* in the number of cities and roads!

Exercise 2.4. Find a linear-time solution to the Travelling Salesman Problem on a DAG.

2.2 DP on Trees: The Longest Path Problem

DAGs are not the only naturally acyclic structures. *Trees* are also another such structure. (Recall that an unrooted tree is just a connected *acyclic* graph.) Thus, we can expect to also be able to solve the Longest Path Problem on trees. In fact, we will soon see that it can be solved in $\mathcal{O}(n)$ time as well.

Now you may be thinking: "Why not just use the recurrence we got for DAGs? Trees are basically special cases of DAGs, right?" Sadly, that's wrong. DAGs are *directed* graphs, while trees are *undirected* graphs. In fact, for all but the very simplest undirected graphs, there's always a cycle among the states, since you can always go back and forth any given edge!

Does this mean we have to again keep track of the set S of already-visited cities?

The answer is Yes if we're dealing with a general undirected graph, but No for trees! Recall that trees don't have cycles, so the only way to visit a node again is to immediately return to it, i.e., backtracking through the same edge you last walked on. Thus, we can avoid revisiting any node by simply remembering the last edge we went through!

This gives us a simpler recurrence than (1). Since there's at most one edge between any two nodes in a tree, remembering the last edge we went through is equivalent to remembering the previous node we were on. Our recurrence is now:

$$opt(u,p) = \max\left(0, \max_{v \in \mathrm{adj}(u)-\{p\}} c(u,v) + opt(v,u)\right).$$

Exercise 2.5. What does u and p represent?

Exercise 2.6. Find an expression for the answer in terms of $opt(u, p)$. *Hint:* You may need to introduce extra states, since at the very beginning, there's doesn't seem to be any appropriate value for p.

Now, what is the time complexity? In general, an easy upper bound of the running time of a DP solution is

$$\text{(number of states)} \times \text{(time to process a state)}. \tag{2}$$

Let's first look at the number of states. Our state is (u, p), where u and p are nodes, so there are n possibilities for each of them, which seems to give us $\mathcal{O}(n^2)$ states. This is already worse than the "$\mathcal{O}(n)$" we claimed at the beginning is possible!

However, although $\mathcal{O}(n^2)$ is a valid upper bound, it isn't quite tight: notice that p is always a neighbor of u, which means (u, p) is always an edge. Since there are $n - 1$ edges in a tree, there are actually just $2(n - 1) = \mathcal{O}(n)$ states! (The 2 comes from counting (u, p) and (p, u) separately.)

Next, what's the time to process a state? Notice that there's a loop across all neighbors of the node u. In the worst case, a node can have $\mathcal{O}(n)$ neighbors, which means the time to process a state can be $\mathcal{O}(n)$, so our solution runs in $\mathcal{O}(n^2)$ in the worst case. This is clearly polynomial-time, which is much faster than our bitmasking solution. However, we mentioned at the beginning that there's an $\mathcal{O}(n)$ solution, so clearly, we have more work to do.

Now, the running time estimate from (2) is only tight if all or most states are processed in roughly equal time. We can get a tighter bound by simply summing the times to process every state, not assuming that they're all processed in roughly equal times:

$$\sum_{\text{every state } S} \text{(time to process state } S\text{)}. \tag{3}$$

What is the time to process state (u, p)? We need to loop across all neighbors of u, which means it takes $\mathcal{O}(\deg(u))$ time to process it, so a tighter upper bound using (3) is

$$\begin{aligned}\sum_{u} \sum_{p \in \text{adj}(u)} \mathcal{O}(\deg(u)) &= \sum_{\text{node } u} \deg(u) \cdot \mathcal{O}(\deg(u)) \\ &= \sum_{\text{node } u} \mathcal{O}(\deg(u)^2).\end{aligned}$$

Unfortunately, this is still $\mathcal{O}(n^2)$ in the worst case if there's a node with degree $\Theta(n)$, such as in a star tree.

The above tells us that our solution isn't really $\mathcal{O}(n)$, but $\Theta(n^2)$. So how do we solve it in $\mathcal{O}(n)$? In fact, I'll leave it to you to solve, but I'll give an important hint, which is actually a pretty powerful technique when dealing with unrooted trees: **root the tree arbitrarily**. In other words, choose some node r (any node) as the "root", and then draw the unrooted tree as a rooted tree with root r. The node r isn't anything special, but the point is that turning the unrooted tree into a rooted tree introduces an additional structure to our graph: after all, a rooted tree is recursively defined as a node together with a list of (smaller) rooted trees, called its children. This is the natural *acyclic* structure that you want to exploit when coming up with the DP solution.

Exercise 2.7. Solve the longest path problem in $\mathcal{O}(n)$ with DP by rooting the tree arbitrarily.

Exercise 2.8. Solve the following problem on an unrooted tree in $\mathcal{O}(n)$: *Find the largest set of nodes such that no two of them are adjacent.*

(This is called the Maximum Independent Set problem. The version of this problem on a general undirected graph is hard in the same way that the Travelling Salesman Problem and the Longest Path Problem are hard: there are no known polynomial-time solutions, but there is no proof that there aren't either, and it's similarly tied to the P vs NP problem.)

2.3 DP and Topological Sort

We've seen in this section that DP is intimately tied with recursion, recurrences, and acyclic structures.

Behind the scenes, what's actually happening with a top-down (memoization) DP is that it's implicitly doing a *topological sort* of the states of the graph. Memoization is the part of the code that prevents any state to be visited more than once.

Bottom-up DP stands in contrast to top-down DP in that you're actually performing a more *explicit* topological sort of the states. For example, when doing a bottom-up DP with the recurrence

$$C(i) = \min(C(i-1)+1, C(i-10)+1, C(i-25)+1),$$

notice that we need to iterate with *increasing* i in order to ensure that the values $C(i-1)$, $C(i-10)$ and $C(i-25)$ have already been computed. Notice that $1, 2, 3, \ldots$ is clearly a toposort of the states.

It's now clear why DP and DAG go together like cookies and cream: DAGs are precisely the directed graphs that can be toposorted. Unrooted trees are not directed so this doesn't apply, but by rooting it and then considering the edges to be directed *downwards*, it becomes a DAG.[4]

[4]The technical term for a "rooted directed tree" is "*arborescence*" which comes from French. You may forget about this term now since no one really uses it (except some computer scientists).

3 More Exponential DPs

3.1 Subsets of Subsets of Subsets of ...

Let's now consider a new problem, which we'll call the **Travelling Salesmen Problem**.

Problem 3.1 (Travelling Salesmen Problem)**.** A snake-oil company wishes to send salesmen to each of n different cities, presumably in order to sell stuff. They have n salesman at their disposal. For every salesman i and every pair of cities u and v, salesman i takes $c(i, u, v)$ nanoseconds to travel from city u to city v. The cost of hiring salesman i is $h(i)$ pesos. Each salesman must start at city 0 (we number the cities 0 to $n-1$), and can only work for a full 8-hour work day, but they may end at any city. The company wants to have each city to be visited by at least one salesman. What is the minimum total cost to do all of this in a single 8-hour work day?

As before, we can precompute the values $w(i, u, v)$, the shortest path from city u to city v for salesman i. You may have learned from a Graphs module that these $\mathcal{O}(n^3)$ values can be precomputed in $\mathcal{O}(n^4)$ time with, say, Floyd's algorithm.

Exercise 3.1. Find an $\mathcal{O}(n^c \cdot n!)$ brute-force solution to this problem.

Just like before, we can improve upon the brute-force solution using DP on subsets, because just like before, it's clear that we don't have to remember the order in which we've visited the previous cities, nor which salesmen did.

3.1.1 Attempt 1

What do we need to remember? Well, there are n salesmen, and they have to be given "travel plans" that are each worth at most 8 hours. Since these salesmen don't interact, we can simply assume that they travel across separate days. There's also no meaning to the order in which they are hired, so we can assume that they're hired in increasing order of index. Thus, at any instant in time, there's only one salesman travelling, say salesman i, and salesmen $1, 2, \ldots, i-1$ cannot be hired anymore. We also need to remember which city u salesman i is currently in, and which set of cities S have yet to be visited by some salesman. Finally, since a salesman can only work for 8 hours, we also need to keep track of the remaining time t salesman i still has available.

This means that our state is (i, u, S, t), where $1 \le i \le n$, $1 \le u \le n$, $S \subseteq V$ and $0 \le t \le$ (8 hours), and we have the following intuitive recurrence for the optimal cost, $opt(i, u, S, t)$.

- If S is empty, i.e., all cities have already been visited, then $opt(i, u, S, t) = 0$.
- Otherwise, $opt(i, u, S, t)$ is the minimum of:
  - $h(j) + opt(j, 0, S - \{0\}, (8\text{ hours}))$ for $i < j \le n$. This represents hiring salesman j next. (What is the 0 for?)
  - $opt(i, v, S - \{v\}, t - w(i, u, v))$ for $v \in S$ such that $w(i, u, v) \le t$, which represents the current salesman, i, visiting city v next. (What's the condition "$w(i, u, v) \le t$" for?)

The answer is then

$$\min_{1\ lei \le n} \left(h(i) + opt(i, 0, S - \{0\}, (8\text{ hours}))\right).$$

Given the meaning of the state (i, u, S, t), it should be clear why the above is correct, so we can use this recurrence as the recurrence of a DP solution.

Exercise 3.2. Explain why there are no cycles among the states.

Now, what's the running time? It's at least the number of states, so how many states are there? Well, i and u can have n values each, and S is a subset of $V = \{0, \ldots, n-1\}$, so there are 2^n values for S. All in all, this gives $n^2 \cdot 2^n$ combinations for (i, u, S). Finally, t is in the interval $[(0 \text{ hours}), (8 \text{ hours})]$, which seems to suggest $n^2 \cdot 2^n \cdot 9$ states...

...However, this is wrong! Remember that the travel times between cities, $c(i, u, v)$ are given in *nanoseconds*, which means it's wrong to assume that t is always a whole number of hours. In fact, it can be any whole number of nanoseconds, and 8 hours is $8 \cdot 60 \cdot 60 \cdot 10^9 \approx 3 \cdot 10^{13}$ nanoseconds. In other words, there are actually $\approx n^2 \cdot 2^n \cdot 3 \cdot 10^{13}$ possible states, which is too many! If the memory limit is 2 gigabytes (roughly $2 \cdot 10^9$ bytes), then there isn't even enough memory to store all the results, let alone time to compute them all!

The main issue here is that we included the remaining time t in our state, but the number of possibilities for just t dwarfs even the number of combinations of the remaining parts of the state—(i, u, S)—which is already exponential in n. So let's try again.

3.1.2 Attempt 2

Let's find a different state, one which doesn't include the time t in our input state. We can't simply drop it, since it's pretty clear that we have to take into account the total travel time in some way. If not in the input, then maybe in the output?

Let's consider a different subproblem. Given a state (i, u, S), what is the minimum travel time needed for salesman i to visit all cities in S starting from city u? Let's denote this by $\mathit{time}(i, u, S)$. This function lets us know which *sets* of cities a salesman can visit within a work day: salesman i can visit all cities in S in a work day iff $\mathit{time}(i, 0, S - \{0\}) \le (8 \text{ hours})$. But now, note that the time is the *output*—it's not anymore part of the input state! So far, so good.

By solving this subproblem, we can now compute the answer with a different recurrence where t doesn't appear in the input state. All we need to remember is i, the lowest-indexed salesman that can still be hired, and the set S of as-yet-unvisited cities. It should be clear now that we have the following recurrence for $\mathit{opt}(i, S)$:

- If S is empty, i.e., all cities have already been visited, then $\mathit{opt}(i, S) = 0$.
- If $i > n$, then $\mathit{opt}(i, S) = \infty$.
- Otherwise, $\mathit{opt}(i, S)$ is the minimum of:
  - $\mathit{opt}(i + 1, S)$. This represents not hiring salesman i.
  - $h(i) + \mathit{opt}(i + 1, S - S')$ for all sets $S' \subseteq S$ such that salesman i can visit all cities in S' in a single work day—equivalently, $\mathit{time}(i, 0, S' - \{0\}) \le (8 \text{ hours})$.

The answer is then simply $\mathit{opt}(1, V)$. Notice that we've cleverly avoided including the time t in any of the input states, either for *opt* or for *time*!

Of course, to complete the solution, we need a recurrence for *time*. We'll leave it to the reader.

Exercise 3.3. Find a bitmask DP solution to compute all of $\mathit{time}(i, u, S)$ in $\mathcal{O}(n^c \cdot 2^n)$ time.

What's the overall running time? There are two functions here, but notice that the *time* values can be computed before any of the *opt* values, and Exercise 3.3 says that this computation takes $\mathcal{O}(n^c \cdot 2^n)$ time for some c. So when computing $\mathit{opt}(i, S)$, we can assume that $\mathit{time}(i, u, S)$ can just be looked up in $\mathcal{O}(1)$.

If we use the looser DP bound (2), then we just need to compute the number of states and the worst-case time to process a single state. The number of states (i, S) is clearly at most $n \cdot 2^n$. Now, to process a single state (i, S), we need to iterate across all *subsets* S' of S. The set S has $2^{|S|}$ subsets, which is 2^n in the worst case, so (2) gives us the upper bound $n \cdot 2^n \cdot \mathcal{O}(2^n) = \mathcal{O}(n \cdot 4^n)$. (This assumes that we represent sets as bitmasks, so all set operations—unions, intersections, set membership, etc.—can be done in $\mathcal{O}(1)$ with bitwise operations.)

Although this is a perfectly fine exponential running time (at least, much better than when t was in the input state!), because of the 4^n factor, this means that we can only feasibly solve the problem for really small n, say $n \le 10$. But what if the constraint for n was "$n \le 12$"?

Let's try to use the tighter bound (3). This is worthwhile since the worst case $\mathcal{O}(2^n)$ to process a state is only ever triggered by a single set, namely $S = \{0, 1, \ldots, n-1\}$, and there are a lot of really small sets S whose processing time $\mathcal{O}(2^{|S|})$ is much smaller than $\mathcal{O}(2^n)$. Indeed, (3) directly gives

$$\sum_{i=0}^{n-1} \sum_{S \subseteq \{0,1,\ldots,n-1\}} \mathcal{O}(2^{|S|}) = n \cdot \sum_{S \subseteq \{0,1,\ldots,n-1\}} \mathcal{O}(2^{|S|}).$$

Thus, we need to simplify the following sum:

$$\sum_{S \subseteq \{0,1,\ldots,n-1\}} \mathcal{O}(2^{|S|}).$$

It turns out that this sum has a pretty nice simple form!

Exercise 3.4. Show that

$$\sum_{S \subseteq \{0,1,\ldots,n-1\}} 2^{|S|} = 3^n.$$

Hint: There are many ways to prove this. For example, you can find a double-counting proof: both sides simply count the number of pairs of sets (S', S) such that $S' \subseteq S \subseteq \{0, \ldots, n-1\}$. Another way is to decompose the left-hand sum according to $|S|$, and noting that there are $\binom{n}{k}$ subsets with $|S| = k$. Finally, you can also prove this by induction, but that's the saddest way.

Therefore, the result of Exercise 3.4 tells us that the running time of this solution is $\mathcal{O}(n \cdot 3^n)$, which is feasible for $n \le 12$ with a reasonable implementation!

However, note that this solution assumes that we can enumerate all "submasks" of a bitmask $\mathcal{O}(2^b)$ time, not $\mathcal{O}(2^n)$ time, where b is the number of "1" bits of the bitmask. It's not exactly obvious, but it turns out to be possible. In fact, there are many ways to do so. You can rediscover some of them in the following exercises.

Exercise 3.5. Show that `u & -u` extracts the least-significant 1 bit of the bitmask `u`.

Exercise 3.6. Use the previous exercise to find a recursive-backtracking solution to enumerate all 2^b submasks of `u`, assuming `u` has b one-bits.

Exercise 3.7. Show that the following code enumerates all submasks of `u`.

```
    int s = 0;
    do cout << s << '\n'; while (s = s - 1 & u);
```

What order does this snippet enumerate all the submasks in?

Exercise 3.8. Show that the following code enumerates all submasks of `u`.

```
    int s = 0;
    do cout << s << '\n'; while (s = s - u & u);
```

What order does this snippet enumerate all the submasks in?

Finally, the following exercise gives you a glimpse of how to compute the running time of a DP if your states and/or transitions involve "subsets of subsets of subsets of ... $\{0, 1, \dots, n-1\}$."

Exercise 3.9. Evaluate

$$\sum_{S\subseteq\{0,\dots,n-1\}} \sum_{T\subseteq S} \sum_{U\subseteq T} c^{|U|}$$

for fixed c and n. (The result is nice.) Then generalize it further.

Hint: If you're stuck, you can also just code this and look at small values; you may figure out a pattern, and then you may be able to come up with a proof.

The sum in the following exercise is more relevant if you're iterating the subsets of $\{0, 1, \dots, n-1\} - S$ rather than S.

Exercise 3.10. Evaluate

$$\sum_{S\subseteq\{0,\dots,n-1\}} 2^{n-|S|}.$$

Can you comment on the result, in light of Exercise 3.4?

3.2 Ternary Masks

In the previous DP, we encountered the number 3^n because of the sum

$$\sum_{S\subseteq\{0,1,\dots,n-1\}} 2^{|S|} = 3^n.$$

Exercise 3.4 tells us that that's simply the number of pairs of sets (S', S) such that $S' \subseteq S \subseteq \{0, \dots, n-1\}$.

In our previous solution, the set S is part of our input state, while the set S' isn't; it only exists during the processing of the input state, specifically when enumerating the subsets of S. However, sometimes it turns out that *both* S and S' are part of the input state. Let's illustrate this with another example:

Problem 3.2. A snake-oil company intends to sell n bottles of snake oil. There are m cities, and r unidirectional roads connecting pairs of cities. The cost of traveling through any road is given. Snake oil bottle i must be picked up at city s_i and must be delivered to a customer located in city t_i. To do this, the company sent a single salesman to do all the deliveries. The salesman can carry any number of bottles at any time without incurring any additional travel cost, and can pick up or drop off any bottle with 0 cost. He must start at city 0 but can end at any city. What is the *minimum* total cost to do so?

We're running out of variations of the name "Travelling Salesman Problem", so let's just call this the **Travelling Errand Boy Problem**.

As before, there should be a straightforward brute-force solution.

Exercise 3.11. Find an $\mathcal{O}(n(m+r)\log r + (2n)!)$ solution to the problem.

Bonus: Improve the $\mathcal{O}((2n)!)$ part by only going through plans that "make sense". Show that the running time becomes $\mathcal{O}(n(m+r)\log r + (2n)!/2^n)$.

Since the $\mathcal{O}(n(m+r)\log r)$ preprocessing step is pretty clear, in what follows, we'll assume that that has already been done, and exclude it from our running time big-$\mathcal{O}$ expressions.

Let's now look for a "subset DP" solution. What should our state contain? That is, what do we need to remember?

Let's look at the perspective of the errand boy. At the moment, he's currently in some city u, carrying some subset S of the bottles, and he's already delivered some other subset T of the bottles. In particular, S and T are disjoint. And at the moment, the errand boy has a choice on what to do next: either to pick up another bottle, or to drop off one of the bottles he's carrying.

Now, there could be many ways to reach this state (u, S, T), since there could have been many sequences in which to pick up and drop off the bottles, but that doesn't matter, since all he needs to know to make the next decision is the state (u, S, T), since S determines which bottles he can drop off, and $V - S - T$ determines the bottles he can still pick up.

All in all, this gives us the following recurrences for $opt(u, S, T)$:

- If $T = V$, then $opt(u, S, T) = 0$, since all bottles have been delivered.
- Otherwise, $opt(u, S, T)$ is the minimum of:
    - $w(u, s_i) + opt(s_i, S \cup \{i\}, T)$ for every bottle i that he hasn't picked up yet, i.e., $i \in V - S - T$,
    - $w(u, t_i) + opt(t_i, S - \{i\}, T \cup \{i\})$ for every bottle $i \in S$ he is currently carrying.

Exercise 3.12. Show that there are no cycles among the states (u, S, T).

The answer is then $opt(0, \emptyset, \emptyset)$, since the errand boy has to start at city 0.

Now, what's the running time? Let's use the looser bound (2) this time; the above tells us that every state is processed in $\mathcal{O}(n)$ time, which is pretty small compared to the number of states (which is clearly exponential), so all states are processed in roughly the same amount of time, and (2) should be pretty good. Thus, what remains is computing the number of states (u, S, T). There are at most $2n + 1 = \mathcal{O}(n)$ possibilities for u (why?), and S and T are subsets of $\{0, \dots, n-1\}$, so it seems that there are $\mathcal{O}(n \cdot 2^n \cdot 2^n) = \mathcal{O}(n \cdot 4^n)$ states. However, notice that S and T are *disjoint*, which means that not all of these $2^n \cdot 2^n$ pairs of sets are valid states!

Exercise 3.13. Show that the number of pairs (S, T) of *disjoint* subsets of $\{0, \dots, n-1\}$ is 3^n.

Exercise 3.13 tells us that there are actually just $\mathcal{O}(n \cdot 3^n)$ states, giving us an $\mathcal{O}(n^2 \cdot 3^n)$ solution! (Again, assuming sets operations are $\mathcal{O}(1)$ due to bitmasking.)

Now, how do we implement this? With memoization, it's pretty straightforward; S and T can both be represented as bitmasks, and the tuples (u, S, T) can be the keys to a *map* data structure that contains memoized results. However, using a map is pretty costly, and we usually want to use normal arrays if we can, since reading from and writing to arrays is faster.

Why does the expression 3^n appear? We can interpret it combinatorially: there are n bottles, and each bottle can only be in one of three possible states:

1. Still not picked up.
2. Picked up but not yet delivered.
3. Has been delivered.

This illustrates pretty clearly why there are only 3^n pairs (S, T): we're simply using the sets S and T to be able to represent the "state" of each bottle (above), and the fact that S and T are disjoint means that there's no "fourth" state of being simultaneously "picked up bot not yet delivered" and "has been delivered".

In fact, we can turn this combinatorial observation into an alternative implementation: rather than encoding S and T as two separate bitmasks, let's just represent the 3^n states directly as a base-3 number! The pair (S, T) corresponds to an integer M between 0 and $3^n - 1$, and each base-3 digit of M represents the state of a bottle. Specifically, if we write $M = b_0 \cdot 3^0 + b_1 \cdot 3^1 + \dots + b_{n-1} \cdot 3^{n-1}$, then

- $b_i = 0$ iff $i \notin S$ and $i \notin T$,
- $b_i = 1$ iff $i \in S$ and $i \notin T$,
- $b_i = 2$ iff $i \notin S$ and $i \in T$.

This makes our representation compact, and in fact, we can simply use an array of length 3^n to represent the memo! (Well actually, we need a 2D array since we also need to include u.) Our state (u, S, T) now becomes (u, M), where $0 \le M < 3^n$. Furthermore, since each bottle can only go from a state to a later state, each digit of m only ever increases, so toposorting the states (u, M) is pretty simple: we simply iterate through M in decreasing order!

This should slightly optimize the reading from and writing to the results memo. However, one downside is that we're now using a base-3 representation, so we can't necessarily use bitwise operations anymore, since those operations are inherently base 2! (If there were 4 states for each bottle instead, then we can again use bitwise operators, since base 4 is just base 2 with pairs of bits grouped together!) However, there's still a way around it, as shown in the following exercise:

Exercise 3.14. Find an $\mathcal{O}(n \cdot 3^n)$ precomputation routine that precomputes all possible "base-3 operations" you'll ever need, such as:

- Given $0 \le i < n$ and $0 \le M < 3^n$, find the ith base-3 digit of M.
- Given $0 \le i < n$ and $0 \le M < 3^n$, increase/decrease the ith base-3 digit of M by 1.
- Etc.

3.3 Digit DP

In fact, we don't have to stop at base 2 or base 3. We can have states encoded in base-b numbers for any b. A common one is $b = 10$, which usually occurs in problems involving the digits of decimal numbers.

Here's a motivating problem.

Problem 3.3. A snake-oil salesman was studying computer science and encountered the following problem: Given two integers L and R such that $0 \le L \le R \le 10^{18}$, find the number of integers in the interval $[L, R]$ whose sum of digits is prime. Can you help him solve it?

We'll call this the **Studying Salesman Problem.**

Well, there isn't really much to be said about this problem that we haven't already seen before, except that the state involves the digits of some number. So rather than explaining the full solution, I'll just give you a few hints on how to solve it.

- You can either enumerate the digits of the number from most to least significant, or vice versa. (Some problems can be solved just as easily by either, and some can be solved more easily with one of them. It's up to you to figure out which one works for this problem.)
- Your state must involve something about the number of digits that have been selected, the sum of digits that have been selected, and the range of numbers that can still be chosen.
- Note that the maximum digit sum is $9 \cdot 18 = 162$, which is pretty small.
- It can help to solve special cases of the problem first, e.g., what if $L = 0$? What if R is a power of 10?
- In fact, $ans(L, R) = ans(0, R) - ans(0, L - 1)$ (why?), so you can actually reduce to the case $L = 0$, which makes things somewhat easier to think about—there is essentially no "lower bound" to the number you're constructing.

Exercise 3.15. Solve the Studying Salesman Problem.

4 Problems

4.1 Coding problems

P1 https://progvar.fun/problemsets/dp-on-subsets

P2 https://progvar.fun/problemsets/dp-on-broken-profile

P3 https://progvar.fun/problemsets/dp-on-digits

P4 https://progvar.fun/problemsets/dp-on-trees

4.2 Non-coding problems

N1 The following is an attempt to enumerate all "submasks" of bitmask `mask`:

```python
def print_submasks(mask, i, sub):
    if (1 << i) > mask:
        print(sub)
    else:
        print_submasks(mask, i + 1, sub)
        if (mask & (1 << i)) != 0:
            print_submasks(mask, i + 1, sub ^ (1 << i))
```

The initial call is `print_submasks(mask, 0, 0)`. Explain the issue with it.

N2 The following is another attempt to enumerate all "submasks" of bitmask `mask`:

```python
def print_submasks(mask, i, sub):
    while (1 << i) <= mask and (mask & (1 << i)) == 0:
        i += 1
    if (1 << i) > mask:
        print(sub)
    else:
        print_submasks(mask, i + 1, sub)
        print_submasks(mask, i + 1, sub ^ (1 << i))
```

The initial call is again `print_submasks(mask, 0, 0)`. Explain the issue with it.

N3 The following is an attempt to enumerate all subsequences of `arr` of size k.

```python
def print_combinations(arr, i, taken, k):
    if i == len(arr):
        if k == 0:
            print(taken)
    else:
        print_combinations(arr, i + 1, taken, k)
        taken.append(arr[i])
        print_combinations(arr, i + 1, taken, k - 1)
        taken.pop()
```

The initial call is `print_combinations(arr, 0, [], k)`. Explain the issue with it.

4.3 More coding problems

C1 Solve the following problem.

> *Given two integers n and m, find the number of positive integers x such that no digit appears more times in x than in n, and x is divisible by m.*
> **Constraints:** $0 \leq n \leq 10^{34}$ *and* $1 \leq m \leq 48$

Hint: The problem may appropriately be called an instance of *bag/multiset DP*.

M1 [4★] **Matrix Sum:** `https://projecteuler.net/problem=345` (use DP)

M2 [4★] **Falling Rocks:** `https://www.codechef.com/problems/CHN15F`

M3 [8★] **Right-hand Rule:** `https://www.codechef.com/problems/MAZETRAV`

M4 [8★] **Triominoes:** `https://projecteuler.net/problem=161`

M5 [8★] **Migrating Ants:** `https://projecteuler.net/problem=393`

M6 [8★] **Dividing into Regions:** `https://www.codechef.com/problems/REGIONS`

M7 [16★] **Tim and BSTs:** `https://www.codechef.com/problems/AMR16A`

M8 [16★] **Eulerian Cycles:** `https://projecteuler.net/problem=289`

M9 [32★] **Beautiful Sandwich:** `https://www.codechef.com/problems/BEAUTY`

M10 [32★] **ATM Queue:** `https://www.codechef.com/problems/CHN16D`