4 Dynamic Programming

Dynamic Programming is a form of recursion. In Computer Science, you have probably heard the tradeoff between Time and Space. There is a trade off between the space complexity and the time complexity of the algorithm.[1] The way I like to think about Dynamic Programming is that we're going to exploit the tradeoff by utilizing the memory to give us a speed advantage when looking at recursion problems. We do this because, in general, "memory is cheap", and we care much more about being as time efficient as possible. We will see later on, however, cases where we care very much about the space efficiency (i.e. streaming algorithms).

Not all recursion problems have such a structure. For example the GCD problem from the previous chapter does not. We will see more examples that don't have a Dynamic Programming structure. Here are the properties you should be looking for when seeing if a problem can be solved with Dynamic Programming.

4.1 Principal Properties

Principal Properties of Dynamic Programming. Almost all Dynamic Programming problems have these two properties:

1. Optimal substructure: The optimal value of the problem can easily be obtained given the optimal values of subproblems. In other words, there is a recursive algorithm for the problem which would be fast if we could just skip the recursive steps.

2. Overlapping subproblems: The subproblems share sub-subproblems. In other words, if you actually ran that naïve recursive algorithm, it would waste a lot of time solving the same problems over and over again.

[1] I actual prefer to think about this as a many-way tradeoff between time complexity, space complexity, parallelism, communication complexity, and (in my opinion most importantly) correctness. This has lead to the introduction of the vast field of randomized and probabilistic algorithms, which are correct in expectation and have small variance. But that is for other classes particularly CS 139 and CS 150.

In other words, your algorithm trying to calculate $f(x)$ might recursively compute $f(y)$ many times. It will be therefore, more efficient to store the value of $f(y)$ once and recall it rather than calculating it again and again. I know that's confusing, so let's look at a couple examples to clear it up.

4.2 Tribonacci Numbers

I'll introduce computing 'tribonacci' numbers as a preliminary example. The tribonacci numbers are defined by $T_0 = 1, T_1 = 1, T_2 = 1$ and $T_k = T_{k-1} + T_{k-2} + T_{k-3}$ for $k \geq 3$.

Let's think about what happens when we calculate T_9. We first need to know T_6, T_7 and T_8. But recognize that calculating T_7 requires calculating T_6 as well since $T_7 = T_4 + T_5 + T_6$. So does T_8. This is the problem of overlapping subproblems. T_6 is going to be calculated 3 times in this problem if done naïvely and in particular if we want to calculate T_k the base cases of T_0, T_1, T_2 are going to be called $\exp(O(k))$ many times.[2] To remedy this, we are going to write down a table of values. Let's assume the word model again.

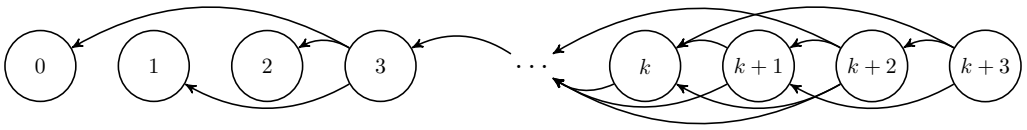

Figure 4.1: Recursion Diagram for Tribonacci Numbers Problem

Algorithm 4.1 (Tribonacci Numbers)**.** Initialize a table $t[j]$ of size k. Fill in $t[0] \leftarrow 1, t[1] \leftarrow 1, t[2] \leftarrow 1$. For $2 \leq j \leq k$, sequentially, fill in $T[j] \leftarrow t[j-1] + t[j-2] + t[j-3]$. Return the value in $t[k]$.

Proof of Correctness. We proceed by induction to argue $t[j] = T_j$. Trivially, the base cases are correct and by the equivalence of definition of $t[j]$ and T_j, each $t[j]$ is filled correctly.[3] Therefore, $t[k] = T_k$.

[2] $\exp(O(k)) = \{f : \exists c > 0, f(k) \leq e^{ck}\}$ where $\leq$ is the relation defined in Section 2. Defining $b = e^c$, then $\exp(O(k)) = \{f : \exists b > 0, f(k) \leq b^k\}$. Therefore, $\exp(O(k)) = \bigcup_{b \in \mathbb{R}^+} O(b^k)$.

[3] Not all proofs of correctness will be this easy, however, they won't be much more complicated either. Aim for 1-2 solid paragraphs. State what each element of the table should equal and argue its correctness.

Complexity. Calculation of each $T[j]$ is constant given the previously filled values. As $O(k)$ such values are calculated, the total complexity is $O(k)$.[4]

As T_j is a monotone increasing sequence in j, we know $T_j \geq 3T_{j-3}$. Therefore, $T_j = \Omega(b^j)$ where $b = \sqrt[3]{3}$. Its not difficult to see that storing $T_1, \ldots, T_k$ then takes $\Omega(b^k)$ space (it's a geometric series). We can make a constant factor improvement on the storage space by noticing that we don't need to store the entire table of $T_1, \ldots, T_k$, we only need to store the last three elements at any given time.

That example was far too easy but a useful starting point for understanding how Dynamic Programming works.

4.3 Generic Algorithm and Runtime Analysis

When you have such a problem on your hands, the generic DP algorithm proceeds as follows:

Algorithm 4.2 (Generic Dynamic Programming Algorithm)**.** For any problem,

1. Iterate through all subproblems, starting from the "smallest" and building up to the "biggest." For each one:
   (a) Find the optimal value, using the previously-computed optimal values to smaller subproblems.
   (b) Record the choices made to obtain this optimal value. (If many smaller subproblems were considered as candidates, record which one was chosen.)
2. At this point, we have the *value* of the optimal solution to this optimization problem (the length of the shortest edit sequence between two strings, the number of activities that can be scheduled, etc.) but we don't have the actual solution itself (the edit sequence, the set of chosen activities, etc.) Go back and use the recorded information to actually reconstruct the optimal solution.

[4] In one of the recitations, we will show how Fibonacci (also same complexity) can actually be run faster than $O(k)$ using repeated squaring.

It's not necessary for "smallest" and "biggest" to refer to literal size. All that matters is that the recursive calls are of "smaller" inputs and that eventually the recursive calls reach the base cases, the "smallest" inputs.

The basic formula for the runtime of a DP algorithm is

$$\text{Runtime} = (\text{Total number of subproblems}) \times \left(\begin{array}{c} \text{Time it takes to solve problems} \\ \text{given solutions to subproblems.} \end{array} \right)$$

Sometimes, a more nuanced analysis is needed. If the necessary subproblems are already calculated, then in the RAM model their lookup is $O(1)$. As our algorithm is to build up the solution, we guarantee that each subproblem is solved, so this $O(1)$ lookup time is correct. Therefore, the total time is bounded by the time complexity to solve a subproblem multiplied by the number of subproblems.

4.4 Edit Distance, an example

Definition 4.3 (Edit Distance)**.** For a given alphabet Σ, an *edit operation* of a string is an insertion or a deletion of a single character. The *edit distance*[5] is the minimum number of edit operations required to convert a string $X = (x_1 \dots x_m)$ to $Y = (y_1 \dots y_n)$.

For example, the edit distance between the words 'car' and 'fair' is 3: 'car' $\to$ 'cair' $\to$ 'air' $\to$ 'fair'. Note the *edit path* here is not unique. The problem is to calculate the edit distance in the shortest time.

The idea for dynamic programming is to somehow recursively break the problem into smaller cases. For convenience of notation, Write X_k to mean $(x_1 \dots x_k)$ (the substring) and similarly $Y_\ell = (y_1 \dots y_\ell)$. The intuition here is the problem of edit distance for $X = X_m$ and $Y = Y_n$ can be broken down into a problem about edit distance of substrings. Formally let $d(k, \ell)$ be the edit distance between X_k and Y_ℓ. The final goal is to calculate $d(m, n)$

[5] This is a actually a metric distance which we define when talking about clustering and packing.

then.

Let's consider what happens to the last character of X_k when calculating $d(k, \ell)$. One of 3 things happens:

(a) It is deleted. In which case the distance is equal to $1 + d(k-1, \ell)$.

(b) It remains in Y, but is no longer the right most character, and so a character was inserted. In which case the distance is $1 + d(k, \ell - 1)$.

(c) It equals the last character of Y, and so it remains the right most character of Y. In which case the distance is equal to $d(k-1, \ell-1)$.

Now, we don't know which of these 3 cases is going to occur. However, we do know that at least 1 of them must be true. Therefore, we can say that the distance is the minimum of these 3 values. Formally:

$$d(k, \ell) \leftarrow \min \begin{cases} d(k, \ell - 1) + 1 \\ d(k-1, \ell) + 1 \\ d(k-1, \ell - 1) + 2 \cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{4.1}$$

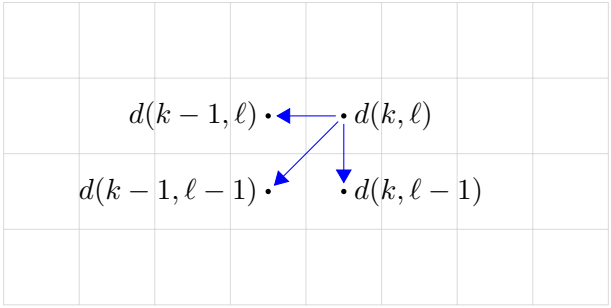

Figure 4.2: Depiction of recursive calls for Algorithm 4.2.

Here $\mathbb{1}_{\{.\}}$ is the indicator function; it equals 1 if the inside statement is true, and 0 otherwise. Therefore, its saying that the distance is 2 if they are not equal (because you'd have to add y_ℓ and remove x_k) and 0 if they are equal (no change required).

The base cases for our problem are $d(k,0) = k$ and $d(0,\ell) = \ell$ (insert or remove all the characters). This gives us all the intuition to formalize our dynamic programming algorithm:

Algorithm 4.4 (Edit Distance)**.** This is a dynamic programming algorithm. Generate a table of size $m \times n$ indexed as $t[k,\ell]$ representing the edit distance of strings X_k and Y_ℓ. Sequentially fill the table in order of increasing k and l by the following rule:

$$t[k,\ell] \leftarrow \min \begin{cases} t[k,\ell-1]+1 \\ t[k-1,\ell]+1 \\ t[k-1,\ell-1]+2\cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{4.2}$$

where any call to $t[k,0] = k$ and $t[0,\ell] = \ell$. When the table is filled, returned $t[m,n]$.

Algorithm Correctness. We verify that the table is filled correctly. For base cases if one string is empty then the minimum update distance is removing or adding all the characters. Inductively, we prove that the rest of the table is filled. If the last characters of the substrings agree, then an edit path is to consider the substrings without the last characters. Alternatively, the only two other edit paths are to add the last character of Y or remove the last character of X. We minimize over this exhaustive set of choices thus proving correctness.

Algorithm Complexity. Each entry of the table requires $O(1)$ calculations to fill given previous entries. As there are $O(mn)$ entries, the total complexity is $O(mn)$ in the word model.

4.5 Memoization vs Bottom-Up

In the previous problem, we generated a table of values t of size $m \times n$ and sequentially filled it up from the base cases to the most complex cases. At any given time, if we tried to calculate item $t[k,\ell]$, we were assured that the subproblems whose value $t[k,\ell]$ is based off of were already solved. In doing so, we had to fill up the entire table only to calculate the value at $t[m,n]$.

This method is the *Bottom-Up* approach. Start from the bottom and work up to calculating all the higher level values. An alternative to the Bottom-Up method is *memoization* (not memorization). If we need to calculate some value of the table $t[i]$ and it depends on values $t[j_1], t[j_2], \ldots, t[j_k]$ then we first check if the value of $t[j_1]$ is known. If it is, we use the value. Otherwise, we store in the computation stack our current position, and then add a new layer in which we calculate $t[j_1]$ recursively, and store its value. This may create more stack layers, itself. Once this stack computation is complete, the stack is removed and we go on to calculating $t[j_2], t[j_3], \ldots, t[j_k]$ one after another. Then we compute $t[i]$ based on the values of $t[j_1], \ldots, t[j_k]$.

Memoization should remind you of generic recursion algorithms with the additional step of first checking if a subproblems solution is already stored in the table and if so using it. More generally, the memoization algorithm looks like this:

Algorithm 4.5 (Memoized Dynamic Programming Algorithm)**.** Assume that $f(i) = g(f(j_1), \ldots, f(j_k))$ for some function g. Construct a table t and fill in all known base cases. Then,

(a) Check if $t[i]$ is already calculated. If so, return $t[i]$.

(b) Otherwise. Recursively calculate $f(j_1), \ldots, f_(j_k)$ and return $g(f(j_1), \ldots, f(j_k))$.

What are the advantages and disadvantages of memoization? First off, in all the problems we've looked at so far, the dependent subproblems have always had smaller indexes than the problem at hand. This is not always the case. A good example is a dynamic programming algorithm on a graph. Even if you number the vertices, you are still going to have all these weird loops because of the edges that you're going to have to deal with. Memoization helps here because you don't need to fill in the table sequentially.

Secondly, in practice memoization can be faster. Occasionally, not all the entries in the table will be filled meaning that we didn't perform unnecessary computations. However, the computation complexity of memoization and bottom-up is in general the same. Remember we're generally looking for worst case complexity. Unless you can argue that more than a constant-factor of subcases are being ignored then you cannot argue a difference in complexity.

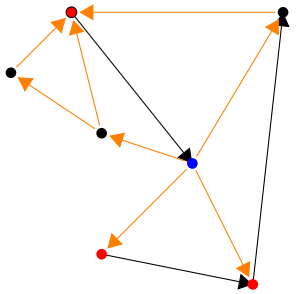

Figure 4.3: Memoization on a directed graph. Assume the red dots are base cases and the orange lines are the recursive calls. The recusion starts at the blue vertex.

Where memoization is worse is the use of the additional stack space. In the case of the Tribonacci numbers, you would generate a stack of size $O(n)$ due to the dependency of each T_j on T_{j-1}. In practice, the computation stack is held in the computer's memory while generally the data that you are manipulating is held on the hard drive. You might not be able to fit a computational stack of size $O(n)$ in the memory.[6] In the case of Tribonacci numbers as every prior Tribonacci number or Edit Distance we can guarantee the entire table will need to be filled. Therefore, bottom-up method is better as it does not use a large stack.

In this class, we won't penalize you for building a bottom-up or memoization dynamic programming algorithm when the other one was better; but it certainly something you should really think about in every algorithm design.

4.6 1D k-Clustering

A classic Dynamic Programming problem is that of 1-dimensional k-Clustering, our next example.

Definition 4.6 (1D Cluster radius)**.** Given a set X of n sorted points $x_1 < \ldots < x_n \in \mathbb{R}$ and set C of k points $c_1, \ldots, c_k \in \mathbb{R}$, the cluster radius is the minimum radius r s.t. every x_i is at most r from some c_j. Quantitatively, this is equivalent to

$$r = \max_{1 \le i \le n} \left(\min_{1 \le j \le k} \left| x_i - c_j \right| \right) \tag{4.3}$$

[6] There are interesting programmatic skirt arounds built for this in many functional languages such as Ocaml or Scala which rely considerably on large recursive algorithms.

Exercise 4.7 (1D k-Clustering)**.** Given a set X of n sorted points in $\mathbb{R}$, find a set C of k points in $\mathbb{R}$ that minimizes the cluster radius.

At first glance, this problem seems really hard and rightfully so. We're looking for k points $c_1, \ldots, c_k$ but they can sit anywhere on the number line. In that regard there are infinite choices that we are minimizing over. Let's simply start with some intuitions about the problem.

When we calculate the cluster radius, we can associate to each point x_i a closest point c_j. So let's define the set $S_j = \{x_i \ : \ c_j \text{ is the closest point of } C\}$. Since all the points lie on the number line, pictorially, these sets partition the number line into distinct regions. Figure 4.4 demonstrates this partition.

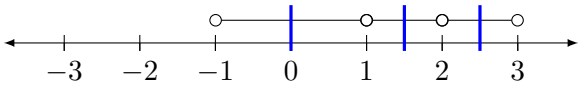

Figure 4.4: Assume that $C = \{-1, 1, 2, 3\}$. This divides the number line into partitions indicated by the blue lines. For any x_i the unique point of C within the partition is the closest point of C. In particular, for any point c_j, S_j is the set of x_i in its partition.

By figuring out this relation to partitions of the number line, we've reduced our problem to finding out which partition of n points minimizes the cluster radius. How many partitions are there? Combinatorics tells us there are $\binom{n-1}{k-1} \in O(n^{k-1})$ partitions. We can do even better.

Our intuition with any dynamic programming algorithm starts with how we state a subproblem. One idea is to define the subproblem as the minimum j-cluster radius for points $\{x_1, \ldots, x_t\}$ for $1 \leq j \leq k$ and $1 \leq t \leq n$. This is a good idea because (a) conditional on items $1, \ldots, t$ forming the union of j clusters, the rest of the solution doesn't matter on how the clustering was done internally and (b) there are $O(nk)$ subproblems.

Algorithm 4.8 (1D k-Clustering)**.** Construct a $n \times k$ table of indexed $r[t, j]$ and another indexed $c[t, j]$. In the second we will store a set of centers of magnitude j. For each $j = 1$

to k and for each $t = 1$ to n: If $j = 1$ then set

$$r[t, j] \leftarrow \frac{x_t - x_1}{2}, \qquad c[t, j] \leftarrow \left\{ \frac{x_t + x_1}{2} \right\} \tag{4.4}$$

Otherwise range s from 1 to $t - 1$ and set s to minimize $f(s) = \max \left\{ r[s, j-1], \frac{x_t - x_{s+1}}{2} \right\}$ and set

$$r[t, j] \leftarrow f(s), \qquad c[t, j] \leftarrow c[s, j-1] \cup \left\{ \frac{x_t + x_{s+1}}{2} \right\} \tag{4.5}$$

Algorithm Correctness. The table r stores the radius for the subproblem and c stores the the optimal set of centers that generate it. In the case that $j = 1$, then we are considering a single center. The optimal center is equidistant from the furthest points which are x_1 and x_t and hence is their midpoint.

Otherwise, we can consider over all possible splitting points s of where the right-most cluster should end. If the next-to-last cluster ends at point x_s, then the cluster radius of the first $j - 1$ clusters is recursively $r[s, j - 1]$, and the the optimal j^{th} cluster center lies at $\frac{x_t + x_{s+1}}{2}$ with cluster radius $\frac{x_t - x_{s+1}}{2}$. The overall radius is the maximum of these two values. Therefore, the optimal clustering of the first t points into j clusters, with the second-to-last cluster ending at point x_s, has radius $f(s)$, with cluster set $r[s, j-1] \cup \left\{ \frac{x_t + x_{s+1}}{2} \right\}$. Minimizing over all possible values of s thus yields the optimal clustering of the first t points into j clusters.

Algorithm Complexity. There are $O(nk)$ subproblems and each subproblems computation requires at most $O(n + k)$ steps, iterating through the possible values of s and then writing down the centers. But as $k \leq n$ (otherwise the problem is trivial), we can lump these two together to say that the subproblem runtime is $O(n)$. Therefore, the overall runtime is $O(n^2 k)$.

Furthermore, notice that $r[s, j]$ will be a (weakly) monotone increasing quantity in s and that similarly $(x_t - x_s)/2$ is monotone decreasing in s. This allows us to instead run a binary search to find the optimal s instead of a linear scan, decreasing the runtime to $O(nk \log n)$ as each subproblem now runs in $O(\log n)$.[7]

[7] In which case, we cannot lump together the writing of the centers in the table and the iterating through possible values of s. The runtime of this is $O(\log n + k)$. However, there is a fix by instead storing a pointer

4.7 How exactly hard is NP-Hard?

Definition 4.9 (Pseudo-polynomial Time)**.** A numeric algorithm runs in *pseudo-polynomial time* if its running time is polynomial in the numeric value of the input but is exponential in the length of the input - the number of bits required to represent it.

This leads to a distinction of NP-complete problems into *weakly* and *strongly* NP-complete problems, where the former is anything that can be proved to have a pseudo-poly. time algorithm and those than can be proven to not have one. NP-hardness has an analogous distinction.

We're going to explore a weakly NP-complete problem next.

4.8 Knapsack

The following is one of the classic examples of an NP-complete problem. We're going to generate a pseudo-polynomial algorithm for it.

Exercise 4.10 (Knapsack)**.** There is a robber attempting to rob a museum with items of positive integer weight $w_1, \ldots, w_n > 0$ and positive integer values $v_1, \ldots, v_n$, respectively. However, the robber can only carry a maximum weight of W out. Find the optimal set of items for the robber to steal.

The brute force solution here is to look at all 2^n possibilities of items to choose and maximize over those whose net weight is $\leq W$. This runs in $O(n2^n \log W)$ and is incredibly inefficient. The $\log W$ term is due to the arithmetic complexity of checking if a set of weights exceeds W.

We apply our classical dynamic programming approach here. Define $S(i, W')$ to be the optimal subset of the items $1, \ldots, i$ that have weight bound W' and their net value and $V(i, W')$ to be their optimal value.

Algorithm 4.11 (Knapsack)**.** Using memoization, calculate $S(n, W)$ according to the following definitions for $S(i, W')$ and $V(i, W')$:

1. If $W' = 0$ then $S(i, W') = \emptyset$ and $V(i, W') = 0$.

to the subproblem from which the optimal solution is built and then backtracking to generate the k centers.

2. If $i = 0$ then $S(i, W') = \emptyset$ and $V(i, W') = 0$.

3. Otherwise, if $V(i-1, W') > V(i-1, W' - w_i) + v_i$

   - then $V(i, W') = V(i-1, W')$ and $S(i, W') = S(i-1, W')$.
   - If not, then $V(i, W') = V(i-1, W'-w_i)+v_i$ and $S(i, W') = S(i-1, W'-w_i) \cup \{i\}$.

Algorithm Correctness. For the base cases, if $W' = 0$ then the weight restriction doesn't allow the selection of any items as all items have positive integer value. If $i = 0$, then no items can be selected from. So both have $V = 0$ and $S = \emptyset$. For the generic problem (i, W') we consider the inclusion and exclusion of item i. If we include i then among the items $1, \ldots, i-1$ their weight cannot exceed $W' - w_i$. Hence their optimal solution is $S(i-1, W'-w_i)$ and the total value is $V(i-1, W'-w_i)+v_i$. If we exclude item i, then the value and solution is the same as the $(i-1, W')$ problem. By maximizing over this choice of inclusion and exclusion, we guarantee correctness as a consequence of the correctness of the subproblems.

Algorithm Complexity. There are $O(nW)$ subproblems and each subproblem takes $O(1)$ time to calculate given subproblems.[8] Therefore a total complexity of $O(nW)$.

We should note that the complexity $O(nW)$ makes this a pseudo-poly time algorithm. As the number W only needs $O(\log W)$ bits to be written, then the solution is exponential in the length of the input to the problem.[9]

4.9 Traveling Salesman Problem

Possibly the most famous of all NP-complete problems, the Traveling Salesman Problem is one of the classic examples of Dynamic Programming.

Exercise 4.12 (Traveling Salesman Problem)**.** Given a complete directed graph with $V = \{1, \ldots, n\}$ and non-negative weights d_{ij} for each edge (i, j), calculate the least-weight cycle that visits each node exactly once (i.e. calculate the least-weight Hamiltonian cycle).

[8] Actually, as I have stated the solution, copying $S(i-1, \cdot)$ into the solution of $S(i, W')$ is time-consuming. Instead, a backtracking solution is required to actually achieve this complexity. But totally doable!

[9] An interesting side note is that all known pseudo-polynomial time algorithms for NP-hard problems are based on dynamic programming.

Naïvely, we can consider all $(n-1)!$ cyclic permutations of the vertices. The runtime is $O(\exp(n \log n))$ by Stirling's approximation.

Our strategy, like always, is to find a good subproblem. For a subset $R \subseteq V$ of the vertices, we can consider the subproblem to be the least path that enters R, visits all the vertices and then leaves. Let's formalize this definition. For $R \subseteq V$ and $j \notin R$, define $C(R, j) =$ cost of cheapest path that leaves 1 for a vertex in R, visits all of R exactly once and no others, and then leaves R for j.

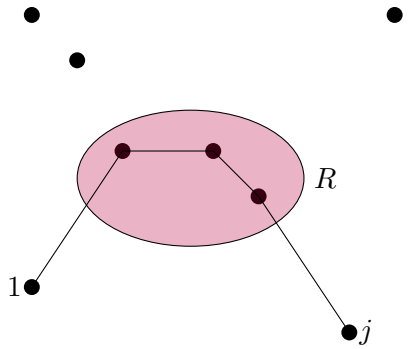

Figure 4.5: Subproblem of the Traveling Salesman Problem

Therefore an equivalent value for the traveling salesman problem is $C(R = \{2, \ldots, n\}, 1)$, as we can start the cycle from any vertex arbitrarily. The recursive solution for C is

$$C(R, j) \leftarrow \min_{i \in R} \left(d_{ij} + C(R - \{i\}, i) \right) \tag{4.6}$$

Let's briefly go over why this is correct: We are optimizing over all $i \in R$ for the node in the path prior to j. The cost would be the cost to go from 1 to all the vertices in $R - \{i\}$ then to i and then to j. That cost is precisely $C(R - \{i\}, i) + d_{ij}$.

Note that any set $R \subseteq V$ can be expressed by a vector of bits of length n (each bit is an indicator) you can show that each subproblem is $O(n)$ time plus the subsubproblem time as we're choosing $i \in R$ and $|R| \leq n$. As there are $O(n \cdot 2^n)$ subproblems, the total time complexity is $O(n^2 2^n)$, a significant improvement on the naïve solution.