Dynamic Programming: Edit Distance, Subset-Sum, and CKY Parsing

Suhas Arehalli

COMP221 - Spring 2025

Oftentimes, we can structure a problem as a *recurrence relation*, where we characterize the solution to one instance of the problem in terms of one or more solutions to *smaller* instances of the problem, where these smaller instances move closer towards some easy to solve base cases.

In the past, we've presented how these kinds of problems can be solved by Divide-and-Conquer algorithms. However, these solutions can sometimes be extraordinarily slow because some recursive calls in distinct recursion paths are solving the exact same problem instance — an instance of wasted work! This situation, often referred to as an instance of **overlapping subproblems**, can be solved by enforcing a rule where every instance of the problem is only solved a single time. The first time you solve a problem, you store the answer in memory so that later on you can simply look up the solution rather than solve that instance again.

This technique of trading off memory to save computation time, called **caching** or **memoization**, is enough to solve the overlapping subproblem problem, but this technique often allows for a very efficient iterative (as opposed to recursive) solution. This design technique — caching and an iterative solution — is called **Dynamic Programming**.

To find a Dynamic Programming solution, I recommend the following steps:

1. Develop a recurrence relation for the problem. It may be helpful to think about the problem in terms of a sequential decision process.
2. Confirm that there are overlapping subproblems. If there are, pursue dynamic programming. Otherwise, a simpler Divide-and-Conquer algorithm will suffice!
3. Design the Dynamic Programming Table (the data structure storing answered to solved subproblems) and determine an iteration order through subproblems that ensures that you can replace all recursive calls with DP table lookups.

The best way to practice this design technique is to study a number of examples. As a result, the rest of this document outlines a number of sample problems from Skiena with the above 3 steps.

1 Edit Distance

We define the **edit distance** between two strings s and t as the minimum number of insertions, deletions, and substitutions needed to transform s to t.

Recurrence Relation

Model the process of determining the edit distance as determining the sequence of edits needed to match each character of the string. To match the first character of t by modifying s, we have a couple of options:

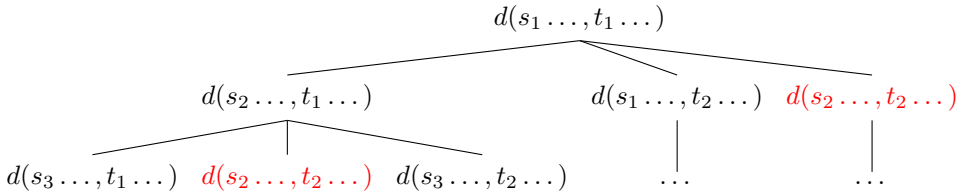
1. If s_1 and t_1 match, then we can simply match them and figure out how to match the rest of the string.
2. Otherwise, we can
   - We can insert t_1 into the first position of s and match the inserted t_1 with t_1 in t.
   - We can delete s_1 and try to match the rest.
   - We can substitute t_1 for s_1 and then match.

Which option is right? We don't select just one! We simply work through each possibility and select the best (minimum) one! That gets us to:

$$d(s_1 \dots s_n, t_1 \dots t_m) = \begin{cases} d(s_2 \dots s_n, t_2 \dots t_m), & s_1 = t_1 \\ 1 + \min(d(s_2 \dots s_n, t_1 \dots t_m), & \\ \qquad d(s_1 \dots s_n, t_2 \dots t_m), & s_1 \neq t_1 \\ \qquad d(s_2 \dots s_n, t_2 \dots t_m)) & \\ \max(n, m) & n = 0 \text{ or } m = 0 \end{cases}$$

Overlapping Subproblems?

Observe the following (truncated) tree expanding the recurrence relation above. One of the overlapping subproblems is highlighted in red.



The DP Table and Iteration Order

Note that the space of relevant problem instances involves selecting the first index of each string. That is, we need solutions to each $d(s_i \dots s_n, t_j \dots t_m)$ for $1 \leq i \leq n+1$ and $1 \leq j \leq m+1$. Thus, we build a table $D[i, j]$ for each of these inputs.

Observe that, from the recurrence relation, we know that in order to solve the instance $d(s_i \dots s_n, t_1 \dots t_m)$, we need to know the solutions to $d(s_{i+1} \dots s_n, t_j \dots t_m)$, $d(s_i \dots s_n, t_{j+1} \dots t_m)$, $d(s_{i+1} \dots s_n, t_{j+1} \dots t_m)$. That is, to fill in $D[i, j]$, we need elements $D[i+1, j]$, $D[i, j+1]$, and $D[i+1, j+1]$ to be filled. We can also fill in $D[n+1, \cdot]$ and $D[\cdot, m+1]$, corresponding to an empty string in either position, using the base case!

This leads us to an algorithm that fills in the table in an order that respects those dependencies. If we iterate through the i's first, then the j's, we get the following pseudocode.

```
function EditDistance(s_1 ... s_n, t_1 ... t_m)
    Let A[i, j] be an array for 1 ≤ i ≤ n + 1 and 1 ≤ j ≤ m + 1
    for i ← 1 to m + 1 do
        A[n + 1, i] ← m + 1 − i
        A[i, m + 1] ← n + 1 − i
    end for
    for i ← n to 1 do
        for j ← m to 1 do
            if s_i = t_j then A[i, j] = A[i + 1, j + 1]
            end if
            A[i, j] ← 1 + min(A[i + 1, j], A[i, j + 1], A[i + 1, j + 1])
        end for
    end for
    return A[1, 1]
end function
```

2 Subset Sum

Given a multi-set $S = \{s_1, \ldots s_n\} \subseteq \mathbb{Z}_+$ of size n and $k \in \mathbb{Z}_+$, subset sum asks you to determine whether there exists a multiset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$.

Recurrence Relation

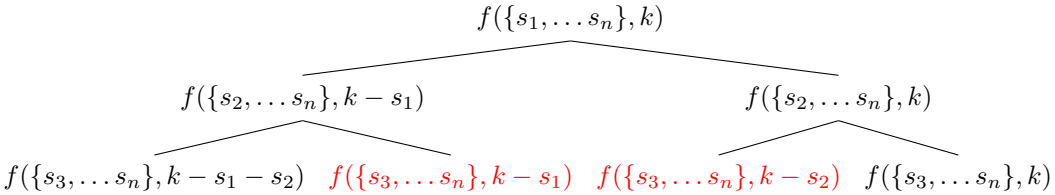
We frame this as a sequential decision problem, where step i involves determining whether s_i should be included in the multi-set. This gets us to the following recurrence relation:

$$f(\{s_1, \ldots s_n\}, k) = \begin{cases} T, & n = 0, k = 0 \\ F, & n = 0, k \neq 0 \\ f(\{s_2, \ldots, s_n\}, k) \mid f(\{s_2, \ldots, s_n\}, k - s_1), & \textit{otherwise} \end{cases}$$

i.e., we either make k or $k - s_1$ with all elements except s_1.

Overlapping Subproblems?

Consider the case where we have a multi-set where $s_1 = s_2$



In this case, we have overlapping subproblems due to possible values of elements in S leading to the same value of the parameter k across different recursive branches. For some small instances, it may be that no elements overlap! However, the fact that subproblems may overlap gives us a

strategy to bound the number of possible values of the second parameter! No matter what the values are, they are bound from above by the initial k and from below by 0 (due to our base cases!).

The DP Table and Iteration Order

A naive solution would build a table A of size $|C| \times (k+1)$ where $A[i, j]$ stores the return value of the call to $f(\{s_i, \dots s_n\}, j)$. We then see a dependency between $f(\{s_i, \dots s_n\}, k)$ and $f(\{s_{i+1}, \dots s_n\}, k)$ and $f(\{s_{i+1}, \dots s_n\}, k - s_i)$. Since we don't know in advance what value s_i will take, we must make sure all possible values of s_i will have the corresponding call cached in advance! Translating this logic to our table, we have that, to fill in $A[i, j]$, we need $A[i+1, j]$ to be filled, as well as $A[i+1, j']$ for all $0 \leq j' < j$. This gets us the following iteration order

function SUBSETSUM($\{s_1, \dots, s_n\}$, k)
  Let $A[i, j]$ be an array for $1 \leq i \leq n+1$ and $0 \leq j \leq k$ with $A[n+1, \dots] \leftarrow F$
  $A[n+1, 0] \leftarrow T$
  for $i \leftarrow n$ to 1 **do**
    for $j \leftarrow k$ to 0 **do**
      if $j - s_i \geq 0$ **then**
        $A[i, j] \leftarrow A[i+1, j] \mid A[i+1, j - s_i]$
      else
        $A[i, j] \leftarrow A[i+1, j]$
      end if
    end for
  end for
  Return $A[1, k]$
end function

Now a clever reader might notice that we don't actually need to store most of the table! Consider the following reduced table!

function SUBSETSUM($\{s_1, \dots, s_n\}$, k)
  Let $A[j]$ be an array for $0 \leq j \leq k$ with $A[\dots] \leftarrow F$
  $A[0] \leftarrow T$
  for $i \leftarrow n$ to 1 **do**
    for $j \leftarrow k$ to 0 **do**
      if $j - s_i \geq 0$ **then**
        $A[j] \leftarrow A[j] \mid A[j - s_i]$
      end if
    end for
  end for
  return $A[k]$
end function

So we get $O(k)$ space rather than $O(nk)$!

3 CKY Parsing

Here, the background is a bit more involved.

A Context-Free Grammar (CFG) consists of a set of rules R, a set of variables V, a set of terminal symbols Σ, and a start symbol $S \in V$, such that a string of terminal symbols $w \in \Sigma^*$ is said to be in the grammar (or *grammatical* w.r.t. that grammar) if we can apply a sequence of rules to go from S to w.

A fun fact about CFGs is that they can be written in *Chomsky Normal Form*, or CNF. This requires that each rule is of one of three forms. We have unary rules, $(A \to \alpha) \in R$, which allows you to replace a variable $A \in V$ with a terminal $\alpha \in \Sigma$, and binary rules, $(A \to BC) \in R$, which allows you to replace a variable A with two variables B followed by C. The third kind of rule allows for empty strings: You can have $(S \to \varepsilon) \in R$, which allows the start symbol to be replaced with the empty string ε.

Critically, we just need to understand the mechanics of parsing with grammars — we can label a terminal symbol $\alpha \in \Sigma$ with the variable $A \in V$ if the rule $(A \to \alpha) \in R$. If we see two adjacent chunks of a string that we've given the labels B and C, we can label the chunk containing both of them with the label A if $(A \to BC) \in R$. The string is in the grammar if we can assign the entire thing the label S. These three ideas let us build a recurrence relation, and subsequently a DP algorithm.

Recurrence Relation

Note here a small strategy we've used to write this recurrence relation. Rather than asking if a string is labeled with an S, we let the potential variable label be a parameter. To allow us to build a recurrence relation, we *generalize* the problem we're trying to solve!

So for a string $w = w_1 \dots w_n$...

$$cky(w_1 \dots w_n, A) = \begin{cases} T, & n = 1, (A \to w_1) \in R \\ F, & n = 1, (A \to w_1) \notin R \\ \bigvee_{1 \le i \le n-1} \bigvee_{(A \to BC) \in R} (cky(w_1 \dots w_i, B) \wedge cky(w_{i+1} \dots w_n, C)), & otherwise \end{cases}$$

This seems tricky, but the *otherwise* case is a fairly straightforward (though not simple!) translation of the logic of parsing to a formula. We ask: Is there a way to split w into two pieces (say, at some index i) such that for some rule $(A \to BC)$ such that the first piece can be parsed as the first symbol in the rule's right-hand side (i.e., $cky(w_1 \dots w_i, B)$) and the second piece can be parsed as the second symbol in the RHS ($cky(w_{i+1} \dots w_n, C)$)? If so, we can parse w as the symbol A. This is the logic of a CFG parsing[1], and we are simply encoding that logic as a formula using logical ors ($\vee$) and ands ($\wedge$).

Overlapping Subproblems?

The DP Table and Iteration Order

Note that in order to cover all possible recursive calls, we need to store the answer to $cky(w_i \dots w_j, A)$ for every choice of start index $1 \le i \le n$, end index $1 \le j \le n$ and variable $A \in R$. This gives us a 3-dimensional table! Here, we conventionally represent the 3D table in two dimensions by simply listing in each cell all values of A for which the call to cky returns T.

[1] Technically, this is the logic of *bottom-up* CFG parsing, one of a couple of parsing strategies for CFGs

From there, we have to think about dependencies between different problem instances to determine the iteration order. To solve the instance $cky(w_i \dots w_j, A)$, we need to know $cky(w_i \dots w_k, B)$ and $cky(w_{k+1} \dots w_j, C)$ for all $i \leq k \leq j-1$ and all $B, C \in V$. These are pretty stringent conditions! Translating that to our table format, for our 2D table, to fill $T[i, j]$ we'd need to know $T[i, k]$ and $T[k + 1, j]$ for all $1 \leq k \leq j - 1$. This requires moving through the table relatively carefully!

One (of many) possible ways to navigate through the table is to observe that the main diagonal ($T[i, i]$ for all i) constitutes our base cases, and by moving one off the diagonal ($T[i, i + 1]$ for reasonable i), we have all of the dependencies we need. Thus, we can slowly move ourselves from the diagonal toward $T[1, n]$, which gives us our answer (if $S \in T[1, n]$, the sentence is grammatical!). Thus we have the following pseudocode:

function CKY($G = (V, \Sigma, R, S)$, $w = w_1 \dots w_n$)
    Let $T[i, j]$ be an array for $0 \leq i, j \leq n$ with $T[\dots, \dots] \leftarrow \emptyset$
    for $i \leftarrow 1$ to n **do**
        for $(A \to \alpha) \in R$ **do**
            if $w_i = \alpha$ **then**
                $T[i, i] \leftarrow T[i, i] \cup \{A\}$
            end if
        end for
    end for
    for $d \leftarrow 1$ to n **do** ▷ This is your offset from the diagonal/length of the substring - 1
        for $i \leftarrow 1$ to $n - d$ **do**
            $j \leftarrow i + d$
            for $k \leftarrow 1$ to $j - 1$ **do**
                for $(A \to BC) \in R$ **do**
                    if $B \in T[i, k]$ and $C \in T[k + 1, j]$ **then**
                        $T[i, j] \leftarrow T[i, j] \cup \{A\}$
                    end if
                end for
            end for
        end for
    end for
    return $S \in A[1, n]$
end function

As far as analysis goes, we can observe that we have an algorithm with time complexity $O(n^3|R|)$ and space complexity $O(n^2|V|)$.