02-613 Week 9

Algorithms and Advanced Data Structures

Aidan Jan

November 3, 2025

Dynamic Programming

Dynamic programming is type of problem that requires reusing (or storing) values or subproblems.

Bellman-Ford is Dynamic Programming

As a review, Bellman-Ford is the algorithm which finds the shortest path between two points, assuming there can be negative edges (and no negative cycles).

We define the function $\texttt{OPT}(v, i)$ as the **min cost** from $S \to v$ in i steps. (S is the start node.)

- If the best $S \to v$ path uses $< i$ edges, then $\texttt{OPT}(v, i) = \texttt{OPT}(v, i-1)$.
- If the best $S \to v$ path uses i edges, for some w, then $\texttt{OPT}(v, i) = d(v, w) + \texttt{OPT}(w, i-1)$

Summarized, we have:

$$\texttt{OPT}(v, i) = \min \begin{cases} \texttt{OPT}(v, i-1) \\ \min_{w \in N(v)} [\texttt{OPT}(w, i-1) + d(v, w)] \end{cases}$$

Essentially, if we can figure out the set of OPTvalues at the previous iteration, we can solve for all the OPTvalues at the current iteration.

- In the worst case, each recursive case takes $\mathrm{O}(n)$. In the worst case for an OPToperation, we have n vertices and n (i) steps. Therefore, the entire algorithm takes $\mathrm{O}(n^3)$.
- However, this is the naive algorithm. Notice that we probably search for the same OPTvalue multiple times, and therefore it can be more efficient if we store the values.
- We can draw a 2D grid, with one axis representing the vertices (v) and one axis representing allowed path length (i).
- Filling the grid (naively) also takes $\mathrm{O}(n^3)$. However, we can also analyze that each update for allowed path length (e.g., one row of the table) only requires iterating through every *edge*. Therefore, we can write the runtime as $\mathrm{O}(VE)$ instead.

Subset Sum

Given an integer bound W and n items, each with weight $w_i \in \mathbb{N}$, find a subset S of items to maximize $\sum_{i \in S} w_i$ while $\sum_{i \in S} w_i < W$. To do this, define $\texttt{OPT}(j, l)$ as the optimal subset considering objects $[1, j]$ with max weight $l \leq w$.

We can define the following recurrence relation:

$$\texttt{OPT}(j, l) = \max \begin{cases} \texttt{OPT}(j-1, l) \\ w_j + \texttt{OPT}(j-1, l-w_j) \end{cases}$$

- We can store all the `OPT`values in a table, and have the our goal be to solve for $\texttt{OPT}(n, w)$, which is the bottom corner.
- To initialize the table, we can assume $\texttt{OPT}(0, l) = 0$ and $\texttt{OPT}(j, 0) = 0$, since they correspond to including zero integers and zero maximum weight, respectively.

Knapsack Problem

Given a bound W, and a collection of items (each with weight w_i and value v_i), find a subset S which maximizes $\sum_{i \in S} v_i$ while keeping $\sum_{i \in S} w_i \leq W$.

The recurrence relation can be defined as

$$\texttt{OPT}(j, l) = \max \begin{cases} \texttt{OPT}(j-1, l) \\ v_j + \texttt{OPT}(j-1, l-w) \end{cases}$$

The base and goal is the same as subset sum.

Edit Distance

Given two strings $A = a_1 a_2 \dots a_n$, $B = b_1 b_2 \dots b_m$, measure some similarity or distance between A and B. One way we can do this is by **edit distance**. Edit distance, $d(a, b)$ is defined as the minimum number of single character edits needed to turn A into B.

For example, let $A = \textit{apple}$, $b = \textit{pear}$. The edit distance between A and B is 4, since

```
apple   (del a)
pple    (p -> e)
pele    (l -> a)
peae    (e -> r)
pear
```

How do we calculate this? Let's consider prefixes. Compare only the first letter of each string at a time. There are only three "moves" we can do: replacement, deletion, insertion. Therefore, we can write the following recurrence relation:

- If A_i `==` B_j,
  - $\texttt{OPT}(i-1, j-1)$ (no change)
- If A_i `!=` B_j,
  - $\texttt{OPT}(i-1, j) + 1$ (insertion)
  - $\texttt{OPT}(i, j-1) + 1$ (deletion)
  - $\texttt{OPT}(i-1, j-1) + 1$ (replacement)

The relation is the minimum of all the above choices. From here, we can draw the classic dp table.

			P	E	A	R
		0	1	2	3	4
	0	0	1	2	3	4
A	1	1	1	2	2	3
P	2	2	1	2	3	3
P	3	3	2	2	3	4
L	4	4	3	3	3	4
E	5	5	4	3	4	4

Gap Scoring

In the context of a genome, we cannot just insert or delete single nucleotides, because if we do, we may possibly mess up a protein (which one amino acid is three base pairs). For example, consider "misspell" versus "mispell".

```
misspell
||| ||||
mis-pell
```

There is one indel, and 7 matches. Now, consider "spite" versus "suite". We can either have one mismatch or two indels.

```
spite     s-pite
|X|||     |  |||
suite     su-ite
```

Depending on what we are optimizing over, it maybe better to have one mismatch, or two indels. Therefore, we now have the notion of a score, which the user can weigh them differently. Compared to edit distance, where every modification is weighted the same, this one does not, and is a more general case.

Sequence Alignment

- Cost of aligning: c
- Cost of indel: λ

$$\text{score}(i,j) = \max \begin{cases} \text{score}(i-1,j-1)+c \\ \text{score}(i,j-1)+\lambda \\ \text{score}(i-1,j)+\lambda \\ \text{score}(i-1,j-1), & \text{only if } x=y \end{cases}$$

Affine Gaps

Affine Gap is a subproblem of sequence alignment. Consider the following two alignments:

```
ATTTGT          ATTTGT
A--TGA          A-T-GA
```

Intuitively, which is better? (It's probably the first one.) Affine gap, instead of just considering the gaps, also consider the rest of the sequence. For example,

- the first alignment here would score (1 mismatch + 3 matches + 1 gap + 2 extensions)
- the second alignment here would serve (1 mismatch + 3 matches + 2 gaps + 2 extensions)

This can be done in O(n^2) using three tables, each table having O(n^2) cells, each taking O(1) to fill.

RNA Folding

RNA is a single strand that folds up

- G and C stick together
- A and U stick together
- Bases closer than 4 together cannot pair
- Pairs cannot cross (e.g., if (i,j) and (k,l) pair, then $i<k<l<j$)

At a given iteration, consider i and j:

```
if we match i + j:
    OPT(i + 1, j - 1) + 1
else if we don't match i:
    OPT(i + 1, j)
else if we match on something in k in [i + 4, j):
    OPT(i + 1, k - 1) + OPT(k + 1, j) + 1
```

With this recurrence relation, we need the base cases:

- The top left corner would be $\texttt{OPT}(i, i + l) \forall l < 4$, which is equal to zero
- We fill from the diagonal, where $i = j$. Because the third case is a loop, we need to start from the diagonal. The above base condition is along the diagonal.
- We only fill half of the entire matrix.

This algorithm has a runtime of $\mathrm{O}(n^3)$. This is because there are $\mathrm{O}(n^2)$ cells to fill in, and each one takes $\mathrm{O}(n)$ time due to the loop.