Dynamic Programming II

Last lecture we introduced the powerful problem-solving paradigm of *dynamic programming*. We saw several example problems all based around variants of the coin change problem. In this part of the lecture, we will study additional examples that look quite different on the surface but follow the same underlying problem-solving pattern.

Dynamic programming is a way of *thinking* about problems that exhibit two key properties:

- **Optimal substructure**: an optimal solution can be built from optimal solutions to smaller instances of the same problem.
- **Overlapping subproblems**: the same smaller problems arise repeatedly.

When these properties are present, dynamic programming often allows us to turn an otherwise exponential-time brute-force algorithm into a polynomial-time one by carefully organizing and reusing work.

Idea: A General DP Design Pattern

Although each dynamic programming problem has its own domain-specific details, we will follow the same high-level process throughout this section:

1. **Reduce the problem to smaller instances.** Ask: *If I make one choice in the solution, what smaller problem remains to be solved?*
2. **Define subproblems precisely.** Identify a set of parameters that uniquely describe the smaller problems. The art here is to define enough subproblems to capture all possibilities, but not so many that their number becomes as inefficient as brute force.
3. **Derive a recurrence relation.** Express the solution to each subproblem in terms of solutions to strictly smaller subproblems, handling base cases explicitly.
4. **Analyze the cost of the solution.** For most problems, the cost is the number of subproblems multiplied by the cost of solving a single subproblem. This is always a valid upper bound. In some cases, we may be able to do a tighter analysis if different subproblems have vastly different costs.
5. **Implement the algorithm in code.** Use either bottom-up evaluation or top-down memoization to ensure that each subproblem is solved only once. If the problem asks for an explicit solution rather than just its value, extract one by storing decisions or by backtracking through the computed subproblems.

The hardest step is almost always defining the *right subproblems*. Once this is done, the recurrence and the implementation usually follow naturally. Today, we will explore some of the different kinds of subproblem design patterns that frequently arise in dynamic programming.

1 Example: The Knapsack Problem

Dynamic programming is especially powerful when a problem involves optimizing a sequence of choices subject to some global constraint. A canonical example, perhaps *the* canonical example of this, is the **knapsack problem**.

Problem: 0–1 Knapsack

We are given n items. Each item i has a *size* s_i and a *value* v_i. We are also given a capacity S. The goal is to select a subset of the items with total size at most S that maximizes total value. Each item may be used *at most* once.

This problem models many real situations: choosing which homework problems to do given limited time, which files to store given limited space, or which tasks to schedule under a fixed resource budget. A brute-force solution would try all 2^n subsets of items, which is infeasible even for moderate n. Dynamic programming will allow us to do much better.

The key observation is that an optimal solution makes a binary choice for each item: *either it is included, or it is not.* Consider some arbitrary item k and suppose we have a knapsack with capacity S. Any optimal solution must fall into one of two cases:

- Item k is *not* used. Then the solution is an optimal solution using the remaining items with capacity S.
- Item k *is* used. Then it contributes value v_k and consumes size s_k, and the remaining problem is an optimal solution using the remaining items with capacity $S - s_k$.

Both of these are just *smaller knapsack problems*, in the first case a smaller set of items with the same capacity, and in the second, a smaller set of items with a smaller capacity, which is exactly what we are looking for!

1.1 Subproblems

The discussion above suggests that a subproblem should record:

- which items we are allowed to consider, and
- how much capacity remains.

Before defining our subproblems, it is worth pausing to consider what *not* to do. At first glance, it appears that we should parameterize on which *subsets* of items to consider. After all, the final solution is a subset of the n items. However, if we define a subproblem for every subset, we would immediately run into trouble: there are 2^n subsets, which is no better than brute force.

This observation highlights a recurring theme in dynamic programming: *the art lies in identifying which distinctions actually matter.* In the knapsack problem, it does not matter which *order* items are considered. All that matters is whether an item is still available or has already been ruled out.

This allows us to impose an arbitrary order on the items, say, items 0 through $n-1$, and then ask, arbitrarily, whether or not we want to use item $n-1$. Whether or not we use item n, we are now left with the subproblem of considering items $\{0,\ldots,n-2\}$. Any subset of items can be represented uniquely by such a sequence of choices. In this sense, dynamic programming replaces an exponential number of subsets with a polynomial number of prefixes.

We therefore define the subproblems for $0 \le k \le n$ and $0 \le B \le S$, as

$$\texttt{MaxVal}(k,B) = \text{maximum value achievable using items } [0,\ldots,k) \text{ with capacity } B.$$

The answer to the original problem is $\texttt{MaxVal}(n,S)$.

Remark: Subproblems should be parameterized by the simplest possible parameters

There is another subtle but important design choice here. One could imagine defining a subproblem whose parameter is an explicit list of remaining items, and then removing one item at each step. While this is conceptually correct, it is algorithmically inconvenient: passing lists around is more expensive, and memoizing such subproblems would require storing and comparing lists.

Instead, we observe that the only information we actually need is *how many* items remain, not their identities. The integer k succinctly encodes the set $[0,\ldots,k)$ and avoids unnecessary overhead. This idea, representing complex state with the simplest possible parameters, is a key skill in designing effective dynamic programs.

1.2 Recurrence Relation

From the optimal substructure argument, we obtain the following recurrence:

Algorithm: Knapsack DP Recurrence

$$\texttt{MaxVal}(k,B) = \begin{cases} 0 & \text{if } k = 0, \\ \texttt{MaxVal}(k-1,B) & \text{if } s_{k-1} > B, \\ \max(\texttt{MaxVal}(k-1,B),\ v_{k-1} + \texttt{MaxVal}(k-1,B-s_{k-1})) & \text{otherwise.} \end{cases}$$

If item k does not fit, we are forced to skip it. Otherwise, we try both possibilities, either taking or skipping the item, and choose the better outcome.

As with other dynamic programming problems, the value computation is straightforward once the recurrence is known, and the selected items can be recovered by backtracking through the subproblems.

1.3 Cost Analysis

There are $(n+1)(S+1)$ distinct subproblems: one for each pair (k,B) with $0 \le k \le n$ and $0 \le B \le S$. Each subproblem is computed using constant work once its dependencies are known.

Therefore, dynamic programming solves the 0–1 knapsack problem in $\Theta(nS)$ time.

1.4 Implementation

We could convert the recurrence relation into a bottom-up implementation like so. We store the values of the subproblems $\texttt{MaxVal}(k, B)$ in a 2D array sequence indexed by the number of items considered and the remaining capacity.

Algorithm: Bottom-up Knapsack Dynamic Program

```
fun knapsack(s : sequence<int>, v : sequence<int>, S : int) -> int:
  n = |s|
  MaxVal = [[0 for _ in 0...S] for _ in 0...n]

  for k in 1...n:
    for B in 1...S:
      if s[k-1] > B:
        MaxVal[k][B] ← MaxVal[k-1][B]
      else:
        MaxVal[k][B] ← max(MaxVal[k-1][B],
                           MaxVal[k-1][B-s[k-1]] + v[k-1])

  return MaxVal[n][S]
```

Remark: Potential Parallelism in the Knapsack Problem

Although we have analyzed the knapsack dynamic program sequentially, its structure exposes a limited but useful amount of parallelism. Note that for this dynamic program, each subproblem $\texttt{MaxVal}(k, B)$ depends **only** on subproblems of the form $\texttt{MaxVal}(k-1, \cdot)$. Therefore, given all the values of $\texttt{MaxVal}(k-1, \cdot)$, one can compute all the values of $\texttt{MaxVal}(k, \cdot)$ independently in parallel.

As a result, the dynamic program can be implemented bottom-up, iterating sequentially across the values of k from 0 to n, and in parallel across the values of B using `tabulate`:

```
for k in 1...n:
  MaxVal[k] ← tabulate((fn (B : int):
    if B == 0: return 0
    else if s[k-1] > B: return MaxVal[k-1][B]
    else: return max(MaxVal[k-1][B], v[k-1] + MaxVal[k-1][B-s[k-1]])
  ), S+1)
```

A purely functional implementation could also be achieved using a fold if desired:

```
base = parallel [0 for _ in 0...S]
MaxVal = fold_left((fn (prev : sequence<int>, k : int):
  // prev represents MaxVal[k-1] => compute MaxVal[k]
  return tabulate((fn (B : int):
    if B == 0: return 0
    else if s[k-1] > B: return prev[B]
    else: return max(prev[B], v[k-1] + prev[B-s[k-1]])
  ), S+1)
), base, 1...n)  // returns MaxVal[n]
```

This results in:

- $\Theta(nS)$ total work, and
- $\Theta(n)$ span, corresponding to the n sequential row dependencies.

In practice, this much parallelism is sufficient to achieve significant speedups, even though the algorithm does not have logarithmic (or polylogarithmic) span.

2 Example: Minimum Edit Distance

The **edit distance** problem provides a formal way to measure how similar two strings are. Intuitively, it asks how many simple edit operations are needed to transform one string into another.

Problem: Edit Distance

Given strings S and T of length n and m respectively, what is the minimum number of **edit operations** required to transform S into T? We allow the following edit operations:

- **Insertion**: insert a character into the string,
- **Deletion**: delete a character from the string,
- **Substitution**: replace one character with another.

For example, the edit distance between `computer` and `commuter` is 1 (substituting `p` with `m`), while the edit distance between `kitten` and `sitting` is 3 (substitute `k` with `s`, substitute `e` with `i`, then insert `g` at the end).

These kinds of problems are extremely useful in fields like computational biology, where, for example, they are used to classify how similar certain DNA sequences are.

2.1 Subproblems

Compared to coin change or knapsack, the subproblems for edit distance are less obvious. The difficulty is not in writing down a recurrence, but in deciding *what information is sufficient* to describe a partial solution. With dynamic programming, it helps to think about what small decisions we can make one at a time to enumerate the possible optimal ways to edit S into T. Each small decision should reduce the problem to a smaller version of itself, the space of which forms our set of subproblems.

Let us think about just the final character of S and T, i.e., $S[n-1]$ and $T[m-1]$. Whatever sequence of edits we apply to S, the final character must inevitably end up equal to $T[m-1]$. There are three ways to make this happen:

1. We could edit the final character, $S[n-1]$, into $T[m-1]$. If $S[n-1]$ already equals $T[m-1]$ then we do nothing, otherwise this is one **substitution**. The problem that remains is to edit $S[0...(n-1)]$ into $T[0...(m-1)]$ in the fewest possible edits

2. We **insert** $T[m-1]$ at the end of S. The problem that remains is to transform $S[0...n)$ into $T[0...(m-1))$ in the fewest possible edits

3. We **delete** $S[n-1]$ and still have to match some earlier character of S with $T[m]$, so the problem remains to transform $S[0...(n-1))$ into $T[0...m)$ in the fewest possible edits

Now we see the important theme. In all three scenarios, the smaller problem we have obtained is just to edit a pair of *prefixes* of the original strings. We therefore define the subproblem

$$\texttt{MED}(i,j) = \text{the edit distance between the prefixes } S[0..i) \text{ and } T[0..j),$$

for all $0 \le i \le n$ and $0 \le j \le m$. The answer to the original problem is then $\texttt{MED}(n,m)$.

2.2 Recurrence Relation

To compute $\texttt{MED}(i,j)$ for $i,j > 0$, we consider how a sequence of edits might make the final characters match. There are three possibilities following the discussion above, which will lead us directly to our recurrence:

1. **Match** or **substitute** the last characters: align $S[i-1]$ with $T[j-1]$, costing 0 if they are equal and 1 otherwise. This reduces the problem to $\texttt{MED}(i-1,j-1)$.

2. **Delete** the last character of S: remove $S[i-1]$, costing 1, and reduce to $\texttt{MED}(i-1,j)$.

3. **Insert** the last character of T: insert $T[j-1]$, costing 1, and reduce to $\texttt{MED}(i,j-1)$.

Since we are looking for the minimum number of edits, we take the *best* of these three choices. Lastly, the base cases will correspond to empty prefixes. To transform an empty string into a string of length j simply costs j insertions, while transforming a non-empty string of length i into an empty one costs i deletions. Putting this all together yields the recurrence

Algorithm: Minimum Edit Distance Recurrence

$$\texttt{MED}(i,j) = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min\left\{ \begin{array}{l} \texttt{MED}(i-1,j-1) + 1_{S[i-1] \neq T[j-1]}, \\ \texttt{MED}(i-1,j) + 1, \\ \texttt{MED}(i,j-1) + 1 \end{array} \right\} & \text{otherwise.} \end{cases}$$

Here, $1_{S[i-1] \neq T[j-1]}$ is an indicator that equals 1 if the characters differ and 0 otherwise.

2.3 Cost Analysis

There are $(n+1)(m+1)$ distinct subproblems, one for each pair (i,j). Each subproblem is computed using a constant amount of work. Therefore, the edit distance dynamic program runs in $\Theta(nm)$ time. As with other dynamic programs, this recurrence can be evaluated bottom-up or top-down with memoization.

2.4 Implementation

A simple sequential imperative implementation of bottom-up edit distance could be written like so. Here, `MED` is again a 2D array sequence indexed by the length of the prefix into S and T.

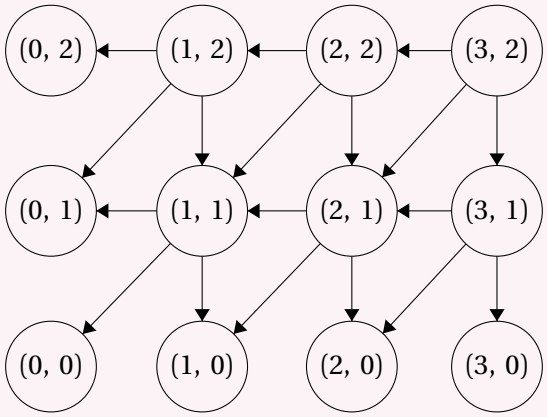
Algorithm: Bottom-Up Edit Distance Dynamic Program

```
fun edit_distance(S : sequence<char>, T : sequence<char>) -> int:
  n = size(S), m = size(T)
  MED = [[i+j for j in 0...m] for i in 0...n]

  for i in 1...n:
    for j in 1...m:
      MED[i][j] ← min(
        MED[i-1][j-1] + (1 if S[i-1] != T[j-1] else 0),
        MED[i-1][j] + 1,
        MED[i][j-1] + 1
      )

  return MED[n][m]
```

Remark: Parallel Evaluation of the Edit Distance Dynamic Program

The dependency structure of the edit distance recurrence is a little more complicated than Knapsack. The subproblems form a 2D grid parameterized by (i, j), but subproblem (i, j) depends on subproblems $(i-1, j-1)$, $(i, j-1)$ and $(i-1, j)$. In other words, it depends on both the previous row *and* previous column. So its not possible to compute all of a row or all of a column in parallel anymore.

However, we could still parallelize a bottom-up implementation of minimum edit distance by noticing that *diagonals* are independent. More specifically, subproblems (i, j) that all share the same value of $i + j$ are independent, so can be computed in parallel.



Therefore, we could implement a bottom-up solution that computes the elements on the each diagonal in parallel. The number of diagonals (i.e., different values of $i + j$) is at most $n + m$. Therefore this can be implemented in $\Theta(nm)$ work and $\Theta(n + m)$ span.

3 Example: Boolean Expression Parenthesisation

We now study a dynamic programming problem whose structure is fundamentally different from the previous examples. Unlike Coin Change or Knapsack, where subproblems were indexed by a numerical parameter or a prefix/suffix of the input, this problem requires reasoning about *intervals* of the input.

Problem: Boolean Expression Feasibility

> You are given a Boolean expression consisting of the literals `True` and `False`, connected by the binary operators `AND` ($\wedge$) and `OR` ($\vee$). The expression is written without parentheses. Is it possible to parenthesize the expression so that it evaluates to `True`?

As with all dynamic programming problems, we begin by asking: *How can we reduce the problem to smaller versions of itself?* Suppose the expression consists of literals

$$b_0 \, \mathtt{op}_0 \, b_1 \, \mathtt{op}_1 \cdots \mathtt{op}_{n-2} \, b_{n-1},$$

where each $b_i \in \{\mathtt{True}, \mathtt{False}\}$ and each $\mathtt{op}_i \in \{\wedge, \vee\}$.

Any fully parenthesized expression must choose *some* operator $\mathtt{op}_k$ as the *last* operation to be evaluated. This splits the expression into two smaller subexpressions:

$$(b_0 \cdots b_k) \, \mathtt{op}_k \, (b_{k+1} \cdots b_{n-1}).$$

Once this split is chosen, we can determine:

- whether or not the left subexpression can evaluate to `True`,
- whether or not the right subexpression can evaluate to `True`,
- whether or not the operator $\mathtt{op}_k$ can combine these values to produce `True`.

This observation suggests that our subproblems should correspond to *contiguous intervals* of the expression.

3.1 Subproblems

We define the following subproblems for $0 \le i < n$ and $i \le j < n$:

$$\mathtt{CanParen}(i, j) = \begin{cases} \textbf{True} & \text{if the subexpression from } b_i \text{ to } b_j \text{ can be} \\ & \text{parenthesized to evaluate to } \mathtt{True}, \\ \textbf{False} & \text{otherwise.} \end{cases}$$

The original problem is therefore to determine the value of $\mathtt{CanParen}(0, n-1)$.

3.2 Deriving the Recurrence

Consider an interval $[i, j]$ with $i < j$. We don't know which of the operators will be the last one applied, so we try *all possible ways* to split the interval at some operator position k, where $i \le k < j$. There are then two cases depending on whether the operator is AND ($\wedge$) or OR ($\vee$).

- If $\texttt{op}_k = \wedge$, then the expression can be `True` only if *both* subexpressions can be `True`.
- If $\texttt{op}_k = \vee$, then the expression can be `True` if *either* subexpression can be `True`.

For the base cases, we consider expressions with just a single value (True or False) and no operators. Clearly, the expression True can be True and the expression False can not be True.

This leads directly to the recurrence:

Algorithm: Boolean Parenthesisation Recurrence

$$\texttt{CanParen}(i,j) = \begin{cases} b_i & \text{if } i = j, \\ \displaystyle\bigvee_{i \le k < j} \begin{cases} \texttt{CanParen}(i,k) \wedge \texttt{CanParen}(k+1,j) & \text{if } \texttt{op}_k = \wedge \\ \texttt{CanParen}(i,k) \vee \texttt{CanParen}(k+1,j) & \text{if } \texttt{op}_k = \vee \end{cases} & \text{otherwise} \end{cases}$$

As in previous dynamic programming problems, this recurrence expresses the solution as a brute-force search over all possible ways to make the final decision, but organized so that each subproblem is solved only once.

3.3 Cost Analysis

There are $\Theta(n^2)$ subproblems, each corresponding to a distinct interval $[i, j]$. For each subproblem/interval, the recurrence tries $(j-i)$ split points in the middle. The value of $(j-i)$ is at most $O(n)$, so the work is at most $O(n^3)$, but with sufficient patience, one can verify that

$$\sum_{0 \le i \le j < n} (j-i) = \Theta(n^3),$$

and hence this is tight. Therefore, with an appropriate bottom-up implementation or top-down with memoization, the total work is $\Theta(n^3)$, and the space usage is $\Theta(n^2)$.

3.4 Implementation

To write a bottom-up implementation we need to establish a valid order to evaluate the subproblems. For interval dynamic programs, we can not simply evaluate $\texttt{CanParen}(i, j)$ in increasing order of i and j like we did for edit distance and knapsack, since this would break the dependencies.

Note that since $\texttt{CanParen}(i, j)$ depends recursively on the intervals (i, k) and $(k+1, j)$, the dependency order can be seen to just be *interval length*. That is, we should evaluate intervals of length 1, then length 2, and so on.

Algorithm: Boolean Parenthesisation Bottom-Up Dynamic Program

```
fun canParenthesize(b : sequence<bool>, op : sequence<char>) -> bool:
  n = size(b)
  CanParen = [[False for _ in 0...n-1] for _ in 0...n-1]

  for i in 0...n-1: CanParen[i][i] ← b[i]

  for length in 0...n-1:
    for i in 0...(n-length)-1:
      j = i + length
      answer = False
      for k in i...j-1:
        if op[k] == '∧':
          answer ← answer or (CanParen[i][k] and CanParen[k+1][j])
        else:  // op[k] == 'v'
          answer ← answer or (CanParen[i][k] or CanParen[k+1][j])
      CanParen[i][j] ← answer

  return CanParen[0][n-1]
```

Remark: Potential Parallelism in Boolean Parenthesisation

Since each subproblem $\texttt{CanParen}(i, j)$ depends only on intervals of smaller length, all of the subproblems of the same length can be evaluated in parallel. Furthermore, the logical or taken over all split points could be expressed as a parallel reduction, so there is quite a lot of parallelisation available in this problem.

The only sequential bottleneck is iterating over the lengths, so by parallelizing over intervals of equal length and doing a parallel reduction, we could evaluate this dynamic program in $O(n^3)$ work and $O(n \log n)$ span.