Dynamic Programming I

Reducing a problem to smaller versions of itself is an extremely common technique in algorithm design. We have already used this technique when studying divide-and-conquer- and contraction-based algorithms. These typically manifest as recursively defined computations. In this lecture we will learn about the third, and arguably most powerful variant of this technique, called *dynamic programming*.

Like divide-and-conquer and contraction, dynamic programming involves reducing a problem to smaller versions of itself. The key difference will be that for dynamic programming problems, we will find ourselves inefficiently solving the *same* smaller problems repeatedly. Addressing this by avoiding the repeated work leads to the technique of dynamic programming.

1 Motivating Example: Coin Change

There are several variants of the coin change problem. We will look at three of them and see how they are all slight variations of the same core theme. For all three variants, dynamic programming will be the key to solving them efficiently.

Problem: Coin Change (Feasibility)

> You are given a description of a currency ($) which has denominations $c_1, \ldots, c_n$ and asked: is it possible to make exactly $\$V$. Denominations may be used multiple times.

We call this the "feasibility" (or "decision") version of the problem because we are just asking a true/false question, whether it possible to make the desired amount. For example, suppose our denominations were $2, 5, 10$. The answer to, "can I make \$6" is true, because I can do $2+2+2$, and for \$12 its also true because I can write $2+10$ or $2+5+5$. On the other hand, the answer is false for \$1 and false for \$3.

1.1 Subproblems

To begin to solve this problem, let us think in terms of reducing it to smaller versions of itself. Continuing with the set of denominations $[2, 5, 10]$, and suppose I was asked if its possible to make a total of \$42. Well, we have a \$2 coin, so if it were possible to make \$40 somehow, then it would also be possible to make \$42. Similarly, if it were possible to make \$37, we could add a \$5 coin to make \$42, and lastly, if we could make \$32 we could add a \$10 coin.

In other words, a way to express the answer to the problem "can I make \$42" is equivalent to: "can I make \$40, or can I make \$37, or can I make \$32". If any of those are true, the answer is true, otherwise its false. This is how, just like in divide-and-conquer and contraction, we have managed to reduce the problem to smaller version(s) of itself! This is the core part, and indeed,

for more complex problems, often the *hardest part* of dynamic programming: we need to define a set of **subproblems**, whose solutions can be used to solve the original problem at hand. So, in our case, given the initial problem of making \$V, we might define the set of subproblems:

$$\texttt{Possible}(v) := \begin{cases} \textbf{True} & \text{if it is possible to make exactly } \$v, \\ \textbf{False} & \text{otherwise.} \end{cases} \qquad \text{for all } 0 \leq v \leq V.$$

1.2 Recurrence Relation

Armed with our subproblems, we need to solve them. As is the case with divide-and-conquer and contraction, our other reduce-to-smaller-version-of-the-problem techniques, this naturally leads to a recursive solution to the problem, since we want to solve the subproblems by combining the answers to smaller subproblems. Before being implemented as code, it is often helpful to first write down the solution in mathematics, as a *recurrence relation*. We have already dealt with recurrence relations heavily in this class, as tools to analyze costs, but that is not their only use. We will also use them to specify solution to dynamic programming problems in a code-independent, language-independent way.

In plain words, we intuited earlier that to solve the problem of making $\$v$, we had to consider the possible denominations $c_1, \ldots, c_n$ and ask whether it was possible to make $\$(v - c_1)$, or make $\$(v - c_2)$, etc. Written as a recurrence relation, our solution therefore ends up looking like

$$\texttt{Possible}(v) = \begin{cases} \textbf{True} & \text{if } v = 0, \\ \bigvee\limits_{\substack{i \in [n] \\ c_i \leq v}} \texttt{Possible}(v - c_i) & \text{otherwise} \end{cases}$$

Remember that the big $\vee$ operator means logical **or**, so it is asking whether any of its operands are true, and if so, the answer is true, otherwise its false. We limit the operands to those c_i such that $c_i \leq v$ to avoid infinite recursion by going negative and missing the base case.

1.3 A recursive algorithm

If we were to translate this recursive solution into code it might look something like this if we write it in a simple imperative style.

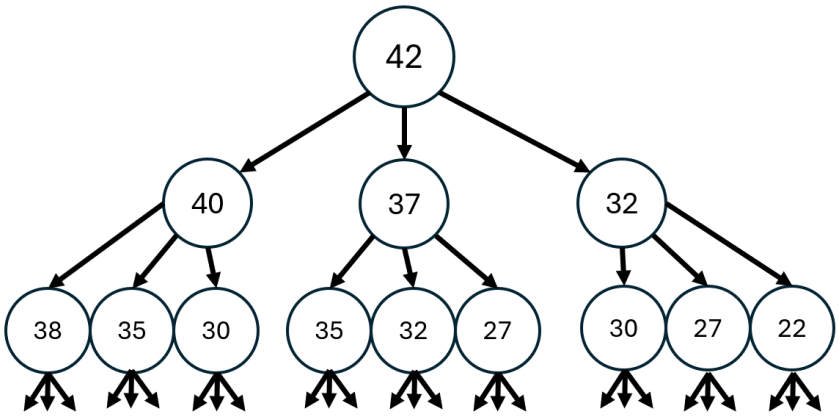
```
fun coin_change(c : sequence<int>, V : int) -> bool:
  fun possible(v):
    if v == 0: return True
    else:
      answer = False
      for denomination in c:
        if denomination <= v:
          answer ← answer or possible(v - denomination)
      return answer
  return possible(V)
```

This inner aggregation could also be written using higher-order functions to be more functional and introduce opportunities for parallelism, but we avoid that here because it obfuscates the recurrence and the dependency structure that we are trying to emphasize, and is less generalizable to other problems.

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  fun possible(v):
    return v == 0 or any(map(fn(x) => (x <= v and possible(v - x)), c))
  return possible(V)
```

`any` is a helper function that returns `True` if any of the elements of the given boolean sequence are `True`. It can be implemented `reduce(or, False, S)`.

Both of these correctly solve the problem, but with one major drawback: their cost is exponential in V. So, even for modest values of V, this code will possibly never terminate in a reasonable amount of time. To see why, we can look at a graph of the recursive calls made by the algorithm. Consider our running example denominations of $2, 5, 10$ and the goal to make \$42. The subproblem for $v = 42$ recursively solves the subproblems for $40, 37, 32$, and so on.



One thing to notice about this tree, right off the bat, is that it is growing exponentially, by a factor of three on each level. In fact, the runtime of evaluating this recurrence is about $\Theta(1.27^V)$, indeed exponential in V.

2 Efficient Evaluation of Dynamic Programs

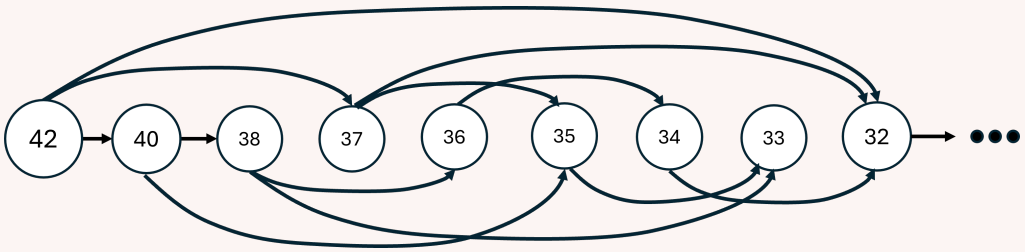
If we take a close look at the tree, we should see a **huge optimization opportunity**. Although the tree is evaluating an exponential number of subproblems, there is huge redundancy among them! We can already see by level three that the subproblem for $v = 30$ is being evaluated on both sides of the tree, and so are the subproblems for $v = 35$ and $v = 32$.

Indeed, the way we defined the subproblems, there are only 43 of them (`Possible`(v) for $0 \leq v \leq 42$), but if we were to try to draw out the entire thing, this tree will contain an insane 34833 subproblems! Our goal is to improve the computation so that each subproblem is computed at most once.

This is the essence of **dynamic programming**. In dynamic programming, we will program simple recursive solutions to problems, which would ordinarily take exponential time due to them needing to repeatedly make the same recursive calls. However, we speed these algorithms up by reusing, or "sharing", the results of our function.

Idea: Dependency graph

The recurrence defines a **dependency graph** in which `Possible`(v) depends only on values `Possible`($v - c_i$), i.e., for smaller values of v. This graph is acyclic and totally ordered by v, which suggests evaluating the subproblems in *increasing order* of v.



There are two ways we can implement dynamic programs. The first one is referred to as **bottom-up** dynamic programs. In bottom-up dynamic programming, we will iteratively computing the results of subproblems in order from smaller to larger, using the smaller ones to find solutions to larger and larger subproblems until we have have enough to solve the original problem.

The other implementation method is called **top-down** dynamic programming, also known as memoization. The core idea for memoization is that we will maintain a cache of results while we compute various results from our recursive function, which stores all the past results that we have computed.

Every time we call our recursive function, we will first check this cache to see if we have already performed this call. If so, we can return the result in constant time. Otherwise, we will compute the result, and also store it in the cache for later use.

Remark: Dynamic Programming as "Clever Brute Force"

One helpful way to think about dynamic programming is as a form of *clever brute force*.

The recursive recurrence above is essentially a brute-force search: to decide whether we can make $\$v$, we try every possible final coin and recursively check whether the remaining amount can be made. This approach explores the entire solution space and applies no heuristics or pruning. As a result, its correctness is immediate: if a solution exists, the recursion will eventually find it.

The inefficiency comes from solving the same subproblems repeatedly. Dynamic programming does not change *what* subproblems are considered; instead, it changes *how* they are evaluated. By ensuring that each subproblem is solved only once, we turn an exponential-time brute-force search into an efficient polynomial-time algorithm.

2.1 Bottom-up evaluation

In a *bottom-up* dynamic programming approach, we solve subproblems in increasing order of size, ensuring that whenever we compute the value of a subproblem, all of the smaller subproblems it depends on have already been solved.

In the coin change problem, the subproblems are indexed by the value v. The recurrence for `Possible`(v) depends only on values `Possible`($v - c_i$) for $c_i \leq v$, which are strictly smaller than v. This means we can safely compute the table in increasing order from 0 up to V.

We maintain an array `possible` where `possible[v]` stores the answer to the subproblem `Possible`(v). Initially, all entries are undefined ($\perp$). Their values are then computed in order of smallest to largest value of v by implementing the recurrence relation.

Algorithm: Bottom-up coin change

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  possible = [⊥ for v in 0...V]

  for v in 0...V:
    if v == 0:
      possible[v] ← True
    else:
      answer = False
      for denomination in c:
        if denomination <= v:
          answer ← answer or possible[v - denomination]
      possible[v] ← answer
  return possible[V]
```

Remark: Computing the final answer

Note that the final answer is stored in `possible[V]`. In general, it is not necessarily the case that the answer to the original problem is exactly one of the subproblems. It might require some further postprocessing to determine the final answer by aggregating many subproblems. We will see some examples of this later in the course.

2.2 Top-down evaluation

An alternative way to evaluate the same recurrence is a *top-down* approach, also known as **memoization**. Instead of explicitly filling in the table from smallest to largest subproblem, we write a recursive function that mirrors the expoential-time recursive algorithm, but with an added *cache* of the results of subproblems as they are computed.

Conceptually, this approach starts by attempting to solve the original problem `Possible`(V) and recursively explores smaller subproblems as needed. Whenever the value of a subproblem is computed for the first time, it is stored in the memoization cache. Subsequent calls reuse the cached result instead of recomputing it.

Algorithm: Top-down coin change

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  memoized = [⊥ for v in 0...V]

  fun possible(v):
    if memoized[v] == ⊥:
      answer = False
      for denomination in c:
        if denomination <= v:
          answer ← answer or possible(v - denomination)
      memoized[v] ← answer
    return memoized[v]

  return possible(V)
```

Remark: Functional vs. Imperative Dynamic Programming

In this lecture, we have presented dynamic programming algorithms using mutable sequences and loops, which makes the flow of computation and the cost analysis clear.

The same algorithms can also be written in a purely functional style (i.e., without mutation). For example, a top-down dynamic program can be implemented as a recursive function that takes and returns both a persistent memo table and result, rather than mutating the table in place. Similarly, bottom-up evaluation can be expressed using a fold operation on a persistent data structure that stores the subproblems.

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  fun possible(table : PersistentDict<int,bool>, v : int)
      -> PersistentDict<int,bool>:

    answer = any(map(fn (x) => (x <= v and
                      find(table, v - x) == SOME(True)), c))
    return insert(table, v,  answer)

  table = fold(possible, {0: True}, 1...V)
  return find(table, V) == SOME(True)
```

The above code implements the bottom-up coin change algorithm using purely functional code. Compared to the imperative implementation however, it:

- requires a substantially more complicated data structure for storing subproblems, e.g.., a persistent balanced binary search tree, instead of an array
- has higher cost since `insert` and `find` with a balanced binary search tree will cost $O(\log V)$ instead of $O(1)$

That said, when implementing dynamic programs in a functional language such as SML, you are free to choose the style you find most natural, as long as your implementation correctly follows the intended recurrence.

3 Cost Analysis

We now analyze the cost of the dynamic programming solutions. For simplicity, we begin with the bottom-up version, which has a straightforward cost analysis. We then explain why the top-down (memoized) version has the same asymptotic cost, even though this is less obvious from the code.

Throughout, let:

- V be the target value,
- n be the number of denominations.

3.1 Bottom-up evaluation

Recall the bottom-up algorithm:

- We compute `possible[v]` for all $v = 0, 1, \ldots, V$.
- For each value v, we iterate over all n denominations and perform constant work per denomination.

Claim: Cost of bottom-up coin change

The cost of evaluating the coin change problem with n coins to make value $\$V$ using the bottom-up strategy is $\Theta(Vn)$.

Proof. There are $V+1$ subproblems. For each subproblem, we scan all n denominations and perform constant work. Therefore, the total work is $\Theta(Vn)$. □

The space usage is also $\Theta(V)$, since we store one boolean value for each subproblem.

The bottom-up strategy makes the analysis rather direct. Even without understand dynamic programming, we see that the algorithm simply has two nested loops, one evaluating the $V+1$ subproblems, and one that spends $O(n)$ time evaluating each particular subproblem.

3.2 Top-down Evaluation (Memoization)

The top-down algorithm looks quite different: it is written as a recursive function and may appear to explore an exponential recursion tree. However, the memoization ensures that this does not actually happen.

Idea: Memoization

Although the recursive function `possible(v)` may be called many times, it is computed at most once for each value of v. The first time `possible(v)` is called, it computes the answer by recursively calling smaller subproblems. The result is then stored in `memoized[v]`. Any future call to `possible(v)` returns immediately in $O(1)$ time.

Claim: Cost of top-down coin change

The cost of evaluating the coin change problem with n coins to make value $\$V$ using the top-down strategy is $\Theta(V\,n)$.

Proof. There are $V+1$ subproblems, one for each value $v \in 0,\ldots,V$. Each subproblem, when computed for the first time, performs $O(n)$ work to iterate over the denominations. All other calls return in constant time. Thus, the total work across the entire execution is $\Theta(V\,n)$. □

Remark: Top-down versus bottom-up

Both approaches solve exactly the same set of subproblems and perform the same total amount of work asymptotically: $\Theta(V\,n)$.

The difference is how the subproblems are evaluated:

- Bottom-up evaluates all subproblems in a fixed order, regardless of whether they are needed. This also means that the programmer needs to correctly determine the order and integrate it into the code.
- Top-down evaluates only the subproblems that are reachable from the original query, and figures out the correct order automatically, while memoization prevents redundant work.

For problems like coin change, where almost all subproblems are eventually needed, the two approaches have essentially identical cost. In other problems, top-down evaluation may avoid computing many unnecessary subproblems and might be more efficient.

4 Reconstructing a Solution

So far, our dynamic programming algorithm only outputs the *value* of the solution: is it possible to make exactly $\$V$? (a boolean value). Often, however, we want more: we want to recover an *actual solution*, such as the specific coins used.

There are two standard techniques for reconstructing a solution from a dynamic program. They differ in their space–time tradeoffs. Both ideas apply equally well to bottom-up and top-down dynamic programming.

4.1 Method 1: Storing Decisions During DP

The most direct approach is to store additional information while computing the values of the subproblems. For each subproblem `Possible`(v), we record *which choice* made the value true. For example, we can store a denomination c_i such that

$$\texttt{Possible}(v - c_i) = \textbf{True}.$$

This can be done by maintaining a second array which stores which decision lead to the result.

Algorithm: Reconstructing the list of coins (decision method)

```
fun get_coins(c : sequence<int>, V : int) -> option<sequence<int>>:
  possible = [False for v in 0...V]
  choice   = [⊥     for v in 0...V]

  for v in 0...V:
    if v == 0:
      possible[v] ← True
    else:
      for denomination in c:
        if denomination <= v and possible[v - denomination]:
          possible[v] ← True
          choice[v] ← denomination

  if not possible[V]: return NONE

  // reconstruct solution
  solution = []
  v = V
  while v > 0:
    denomination = choice[v]
    solution ← solution + [denomination]
    v ← v - denomination

  return SOME(solution)
```

This uses $\Theta(V)$ additional space and time, and hence doesn't affect the overall asymptotic cost.

4.2 Method 2: Backtracking

An alternative approach is to reconstruct a solution *after the fact*, using only the values of the subproblems themselves. Suppose we have already computed the array $\texttt{Possible}(v)$. To reconstruct a solution for value v, we search for a denomination $c_i \le v$ such that

$$\texttt{Possible}(v - c_i) = \textbf{True}.$$

Once we find such a coin, we include it in the solution and continue with $v - c_i$ remaining.

Algorithm: Reconstructing the list of coins (backtracking method)

```
fun get_coins(c : sequence<int>, V : int) -> option<sequence<int>>:
  possible = [⊥ for v in 0...V]

  for v in 0...V:
    if v == 0:
      possible[v] ← True
    else:
      possible[v] ← False
      for denomination in c:
```

```
        if denomination <= v:
          possible[v] ← possible[v] or possible[v - denomination]

  if not possible[V]: return NONE

  // reconstruct solution
  solution = []
  v = V
  while v > 0:
    for denomination in c:
      if denomination <= v and possible[v - denomination]:
        solution ← solution + [denomination]
        v ← v - denomination

  return SOME(solution)
```

This method avoids storing extra information, but reconstruction may take more time since it has to essentially reevaluate the recurrence again.

Remark: Storing decisions versus backtracking

Both versions compute the same subproblems. The difference lies entirely in *when* we decide which choices matter:

- Store decisions early: more space, faster reconstruction.
- Recompute decisions later: less space, slower reconstruction.

This illustrates a recurring theme in dynamic programming: *the subproblems encode all solutions implicitly, and reconstruction is simply a matter of extracting one of them.*

5 Counting and Optimisation Problems

Dynamic programming is predominantly used to solve three kinds of problems: feasibility, counting, and optimization. Most (but not all) algorithmic problems fall into one of these three categories. The overall idea is exactly the same:

- Define a suitable set of subproblems
- Write a recurrence relation which solves a subproblem in terms of smaller subproblems
- Analyze the cost by determining the number of subproblems and the cost per subproblem

The power of dynamic programming lies primarily in choosing the right subproblems and writing the correct recurrence. Once these are established, the implementation, cost analysis, and reconstruction techniques follow the same patterns as before. For this reason, in the following variants we focus on the subproblem definitions and recurrences, and omit repeated code.

5.1 The optimization problem: fewest possible coins

Here, we consider an optimization version of the problem.

Problem: Coin Change (Optimization)

Given denominations $c_1, \ldots, c_n$ in some currency (\$), **what is the fewest number of coins** needed to make exactly $\$V$?

As before, we approach this problem by reducing it to smaller subproblems. The key difference is that instead of asking whether a solution exists, or counting all possible solutions, we are now trying to find the *best* solution according to some objective.

To make exactly $\$v$, we first choose a coin to use. If we choose denomination c_i, then we must make the remaining amount $\$(v - c_i)$ somehow. How should this amount be made? Of course, using *the fewest possible coins*, which is just a smaller version of the original problem! We therefore define:

$$\texttt{MinCoins}(v) = \text{minimum number of coins needed to make } \$v.$$

If we decide to include a copy of the coin c_i, we use

$$1 + \texttt{MinCoins}(v - c_i)$$

coins in total. Different choices of the first coin may lead to solutions using different numbers of coins. Since we want the *fewest* coins overall, we should brute-force all possible coins c_i such that $c_i \leq v$ and take the one that minimizes the total number of coins. This leads directly to the following recurrence.

$$\texttt{MinCoins}(v) = \begin{cases} 0 & \text{if } v = 0, \\ \infty & \text{if } c_i > v \text{ for all } i, \\ \min\limits_{\substack{i \in [n] \\ c_i \leq v}} \big(1 + \texttt{MinCoins}(v - c_i)\big) & \text{otherwise.} \end{cases}$$

Note that if no coins can be used to make v, we define $\texttt{MinCoins}(v) = \infty$. This ensures that it will never be selected as the minimum. This is a common practice for optimization-based dynamic programming algorithms where there might not be a valid solution at all. The algorithm again runs in $\Theta(V \cdot n)$ time and $\Theta(V)$ space.

5.2 The counting problem: how many ways?

Lastly, we now consider a counting variant of the coin change problem.

Problem: Coin Change (Counting)

Given denominations $c_1, \ldots, c_n$ in some currency (\$), **how many distinct ways** are there to make exactly $\$V$? We count *ordered* sequences of coins, by which we mean that using denominations $\{2,5\}$, the solutions $2+5$ and $5+2$ are counted as distinct.

The key idea is the same. To make \$42, if I have a \$2 coin, then I can make \$40 and add a \$2 coin. If I have a \$5 coin then I can make \$37 and add a \$5 coin. We therefore define subproblems similarly, in terms of smaller values of v

$$\texttt{CountWays}(v) = \text{number of ways to make } \$v, \qquad \text{for all } 0 \le v \le V.$$

To count the number of ways to make \$42, we can case on the last coin used, say \$2, \$5, or \$10, and consider the number of ways to make \$40, \$37, or \$32 respectively. Each of these give a valid way to make \$42, so we simply *add* the number of ways to make these amounts. The recurrence therefore mirrors the feasibility version, but replaces logical **or** with addition:

$$\texttt{CountWays}(v) = \begin{cases} 1 & \text{if } v = 0, \\ \displaystyle\sum_{\substack{i \in [n] \\ c_i \le v}} \texttt{CountWays}(v - c_i) & \text{otherwise.} \end{cases}$$

The cost analysis is identical to the feasibility version: $\Theta(V \cdot n)$ time and $\Theta(V)$ space.

Remark: The structure of dynamic programming

As with the previous variants, the structure of the dynamic program is the same: we reduce the problem to smaller values of v and combine the results using an appropriate operation: **and**/**or** for feasibility, **sum** for counting, and **min**/**max** for optimization.

Remark: Difficulty of feasibility versus optimisation versus counting

In general, feasibility problems are the easiest to solve, as we only have to determine whether there exists some valid solution at all, not find the best one. It is strictly easier than an optimization or counting problem since, if we could solve the optimisation problem, we could immediately determine feasibility by checking that the answer is not ∞ or $-\infty$, and of course if we can solve the counting problem, we could check feasibility by checking whether or not the answer is zero.

Similarly, optimization problems are typically easier than counting problems. Both generally involving searching the entire solution space, where one finds the best solution and the other adds up how many. The primary difference is that it doesn't hurt an optimization problem to consider a solution multiple times, but for a counting problem, we must be careful to avoid double counting or it changes the answer. So, generally:

Feasibility $\ll$ **Optimization** $\ll$ **Counting**