Dynamic Programming I

Dynamic Programming is a powerful technique that often allows you to solve problems that seem like they should take exponential time in polynomial time. Sometimes it allows you to solve exponential time problems in slightly better exponential time. It is most often used in combinatorial problems, like optimization (find the minimum or maximum weight way of doing something) or counting problems (count how many ways you can do something). We will review this technique and present a few key examples.

Objectives of this lecture

In this lecture, we will:

- Review and understand the fundamental ideas of Dynamic Programming.
- Study several example problems:
  - The Knapsack Problem
  - Independent Sets on Trees (Tree DP)
  - The Longest Increasing Subsequence Problem (SegTree DP)

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 15/14 (3^{rd}/4^{th} ed.), Dynamic Programming
- DPV, *Algorithms*, Chapter 6, Dynamic Programming
- Erikson, *Algorithms*, Chapter 3, Dynamic Programming

1 Introduction

Dynamic Programming is a powerful technique that can be used to solve many combinatorial problems in polynomial time for which a naive approach would take exponential time. Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer", except that the subproblems will *overlap*.

You may have seen the idea of dynamic programming from your previous courses, but we will take a step back and review it in detail rather than diving straight into problems just in case you have not, or if you have and have completely forgotten!

Key Idea: Dynamic programming

Dynamic programming involves formulating a problem as a set of *subproblems*, expressing the solution to the problem *recursively* in terms of those subproblems and solving the recursion *without repeating* the same subproblem twice.

The two key sub-ideas that make DP work are ***memoization*** (don't repeat yourself) and ***optimal substructure***. Memoization means that we should never try to compute the solution to the same subproblem twice. Instead, we should store the solutions to previously computed subproblems, and look them up if we need them again.

1.1 Warmup: Climbing Steps

Lets start with a nice problem to break down these key ideas. Suppose you can jump up the stairs in 1-step or 2-step increments. How many ways are there to jump up n stairs?

Where is the *substructure* in this problem? Well, with n stairs, we have two *choices*, either we jump up 1 or 2 steps. After jumping up 1 step, we will have $n-1$ steps remaining, or after jumping up 2 steps we will have $n-2$ steps remaining. So it sounds like some sensible smaller problems to consider would be the number of ways to jump up n steps for any smaller value of n. Lets define a function *stairs* which counts exactly this. We can evaluate stairs *recursively*

```
function stairs(n : int) -> int = {
  if (n <= 1) return 1;
  else {
    waysToTake1Step = stairs(n-1);
    waysToTake2Steps = stairs(n-2);
    return waysToTake1Step + waysToTake2Steps;
  }
}
```

We found the substructure in the problem, but we're not done yet. Implemented as such, the above code would perform exponentially many recursive calls because it would end up **repeatedly evaluating the same problem**. Notice that stairs(n) calls stairs($n-1$) and stairs($n-2$). But stairs($n-1$) *also* calls stairs($n-2$), so we will call that twice. Going deeper into the recursion, we will see that we compute the same values exponentially many times!

To rectify this, we apply the other key idea of dynamic programming, which is don't repeat yourself, aka. **memoization**. Lets store a lookup table of previously computed values, and instead of recomputing from scratch every time, we will just reuse values that already exist in the table! By convention we will refer to the lookup/memoization table as "memo". The most generic way to implement the memo table is to use a *dictionary* that maps subproblems to their values.

```
dictionary<int, int> memo;

function stairs(n : int) -> int = {
  if (n <= 1) return 1;
  if (memo[n] == None) {
    waysToTake1Step = stairs(n-1);
    waysToTake2Steps = stairs(n-2);
    memo[n] = waysToTake1Step + waysToTake2Steps;
  }
  return memo[n];
}
```

Note that for very many problems, the memo table does not need to be implemented as a hashtable dictionary. In the majority of problems, including this one, the subproblems are just identified by integers from $0 \ldots n$, so the dictionary can actually just be implemented as *an array*! A hashtable dictionary would only be required if the subproblem identifiers can not easily be mapped to a set of small integers.

1.2 The "recipe"

With these key ideas in mind, lets give a high-level recipe for dynamic programming (DP). A high-level solution to a dynamic programming problem usually consists of the following steps:

1. **Identify the set of subproblems** You should **clearly and unambiguously** define the set of subproblems that will make up your DP algorithm. These subproblems must exhibit some kind of *optimal substructure* property. The smaller ones should help to solve the larger ones. This is often the **hardest part** of a DP problem, since locating the optimal substructure can be tricky.

2. **Identify the relationship between subproblems** This usually takes the form of a *recurrence relation*. Given a subproblem, you need to be able to solve it by combining the solutions to some set of smaller subproblems, or solve it directly if it is a *base case*. You should also make sure you are able to solve the original problem in terms of the subproblems (it may just be one of them)!

3. **Analyze the required runtime** The runtime is **usually** the number of subproblems multiplied by the time required to process each subproblem. In uncommon cases, it can be less if you can prove that some subproblems can be solved faster than others, or sometimes it may be more if you can't look up subproblems in constant time.

This is just a *high-level* approach to using dynamic programming. There are more details that we need to account for if we actually want to implement the algorithm. Sometimes we are satisfied with just the high-level solution and won't go further. Sometimes we will want to go down to the details. These include:

4. **Selecting a data structure to store subproblems** The vast majority of the time, our subproblems can be identified by an integer, or a tuple of integers, in which case we can store our subproblem solutions in an array or multidimensional array. If things are more complicated, we may wish to store our subproblem solutions in a hashtable or balanced binary search tree.

5. **Choose between a *bottom-up* or *top-down* implementation** A bottom-up implementation needs to figure out an appropriate *dependency order* in which to evaluate the subproblems. That is, whenever we are solving a particular subproblem, whatever it depends on must have already been computed and stored. For a top-down algorithm, this isn't necessary, and recursion takes care of the ordering for us.

6. **Write the algorithm** For a bottom-up implementation, this usually consists of (possibly nested) for loops that evaluate the recurrence in the appropriate dependency order. For a top-down implementation, this involves writing a recursive algorithm with *memoization*.

2 The Knapsack Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are:

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

Say you have a total of 15 hours: which parts should you do? If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points-per-hour (a greedy strategy). But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?[1]

The above is an instance of the *knapsack problem*, formally defined as follows:

Definition: The Knapsack Problem

We are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).

We can solve the knapsack problem in exponential time by trying all possible subsets. With Dynamic Programming, we can reduce this to time $O(nS)$. Lets go through our recipe book for dynamic programming and see how we can solve this.

[1] Answer: In this case, the optimal strategy is to do parts A, B, F, and G for a total of 34 points. Notice that this doesn't include doing part C which has the most points/hour!

Step 1: Identify some optimal substructure Lets imagine we have some instance of the knapsack problem, such as our example $\{A, B, C, D, E, F, G\}$ above with total size capacity $S = 15$. Here's a seemingly useless but actually very useful observation: The optimal solution either does contain G or it does not contain G. How does this help us? Well, suppose it does contain G, then what does the rest of the optimal solution look like? It can't contain G since we've already used it, and it has capacity $S' = S - s_G$. What does the optimal solution to this remainder look like? Well, by similar logic to before, it must be the *optimal solution* to a knapsack problem of total capacity S' on the set of items not including G! (Formally we could prove this by contradiction again—if there was a more optimal knapsack solution for capacity S' without G, we could use it to improve our solution.) This is another case of *optimal substructure* appearing!

Step 2: Defining our subproblems Now that we've observed some optimal substructure, lets try to define some subproblems. Our observation seems to suggest that the subproblems should involve considering a *smaller capacity*, and considering one fewer item. How should we keep track of which items we are allowed to use? We could define a subproblem for every subset of the input items, but then we would have $\Omega(2^n)$ subproblems, and that's no better than brute force! But here's another observation: it doesn't really matter what order we consider inserting the items if for every single item we either use it or don't use it, so we can instead just consider subproblems where we are using items $1 \dots i$ for $0 \leq i \leq n$. Combining these two ideas, both the capacity reduction and the subset of items, we define our subproblems as:

$V(k, B) =$ The value of the best subset of items from $\{1, 2, \dots, k\}$ that uses at most B space

The solution to the original problem is the subproblem $V(n, S)$.

Step 3: Deriving a recurrence Now that we have our subproblems, we can use our substructure observation to make a recurrence. If we choose to include item k, then our knapsack has $B - s_k$ space remaining, and we can no longer use item k, so this gives us

$$v(k, B) = v_k + V(k-1, B - s_k) \qquad \text{if we take item } k$$

Otherwise, if we don't take item k, then we get

$$v(k, B) = V(k-1, B) \qquad \text{if we don't take item } k$$

Finally, we need some base case(s). Well, if we have no items left to use $k = 0$, that seems like a good base case because we know the answer is zero! So, putting this together, we can write the recurrence:

Algorithm: Dynamic programming recurrence for Knapsack

$$V(k, B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1, B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B - s_k), V(k-1, B)\} & \text{otherwise} \end{cases}$$

Note here that we had to check whether $s_k > B$. In this case, we can't choose item k even if we wanted to because it doesn't fit in the knapsack, so we are forced to skip it. Otherwise, if it fits, we try both options of taking item k or not taking item k, then use the best of the two choices.

Step 4: Analysis We have $O(nS)$ subproblems and each of them requires a constant amount of work to evaluate for the first time. So, using dynamic programming, we can implement this solution in $O(nS)$ time.

A top-down implementation This can be turned into a recursive algorithm. Naively this again would take exponential time. But, since there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. Let us initialize a 2D memoization table `memo[k][b]` to "unknown" for all $0 \le k \le n$ and $0 \le b \le S$.

```
function V(k : int, B : int) -> int = {
  if (k == 0) return 0;
  if (memo[k][B] != unknown) return memo[k][B];     // <- added this
  if (s_k > B) result := V(k-1,B);
  else result := max{v_k + V(k-1, B-s_k), V(k-1, B)};
  memo[k][B] = result;                              // <- and this
  return result;
}
```

Since any given pair of arguments to V can pass through the memo check only once, and in doing so produces at most two recursive calls, we have at most $2n(S+1)$ recursive calls total, and the total time is $O(nS)$.

So far we have only discussed computing the *value* of the optimal solution. How can we get the items? As usual for Dynamic Programming, we can do this by just working backwards: if `memo[k][B]` = `memo[k-1][B]` then we *didn't* use the kth item so we just recursively work backwards from `memo[k-1][B]`. Otherwise, we *did* use that item, so we just output the kth item and recursively work backwards from `memo[k-1][B-s_k]`. One can also do bottom-up Dynamic Programming.

3 Max-Weight Indep. Sets on Trees (Tree DP)

Given a graph G with vertices V and edges E, an *independent set* is a subset of vertices $S \subseteq V$ such that none of the vertices are adjacent (i.e., none of the edges have both of their endpoints in S). If each vertex v has a non-negative weight w_v, the goal of the *Max-Weight Independent Set* (MWIS) problem is to find an independent set with the maximum weight. We now give a Dynamic Programming solution for the case when the graph is a tree. Let us assume that the tree is rooted at some vertex r, which defines a notion of parents/children (and ancestors/descendents) for the tree. Lets go through our usual motions.

Step 1: Identify some optimal substructure Suppose we choose to include r (the root) in the independent set. What does this say about the rest of the solution? By definition, it means that the children of the root are not allowed to be in the set. Anything else though is fair game. In particular, for every *granchild* of the root, we would like to build a max-weight independent set rooted at that vertex. A proof by contradiction would as usual verify that this has optimal substructure.

On the other hand, if we choose to not include r in our independent set, then all of the children are valid candidates to include. Specifically, we would like to construct a max-weight independent set in all of the subtrees rooted at the children (which may or may not contain those children themselves).

Step 2: Define our subproblems The optimal substructure suggests that our subproblems should be based on *particular subtrees*. This general technique is often referred to as "tree DP" for this reason. Our set of subproblems might therefore be

$$W(v) = \text{the max weight independent set of the subtree rooted at } v$$

The solution to the original problem is $W(r)$.

Step 3: Deriving a recurrence Like many of our previous algorithms, we build the recurrence by casing on possible decisions we can make. Keeping with the spirit of that, it seems like the decision we can make at any given problem $W(v)$ is whether or not to include the root vertex v in the independent set. If we choose to not include it, then we should just recursively find a max-weight set in the children's subtrees. We let $C(v)$ be the set of children of vertex v. Then we have

$$W(v) = \sum_{u \in C(v)} W(u) \qquad \text{if we don't choose } v.$$

Suppose we do choose v, then what can we do? As discussed, we can no longer include any of v's children without violating the rules, but we can consider any of v's grandchildren and their subtrees. Let $GC(v)$ denote the set of v's grandchildren. Then we have

$$W(v) = w_v + \sum_{u \in GC(v)} W(u) \qquad \text{if we choose } v.$$

Finally, what about base cases? If v is a leaf then the max-weight independent set just contains v for sure. To write the full recurrence, we just take the best of the two choices, choose v or don't choose v.

Algorithm: Dynamic programming recurrence for max-weight independent set on a tree

$$W(v) = \max\left\{ \sum_{u \in C(v)} W(u), \quad w_v + \sum_{u \in GC(v)} W(u) \right\}$$

Wait, stop! Where's the base case? We must have forgotten it. Or did we? Suppose v is a leaf. Then both of the sums in the recurrence are empty because $C(v)$ and $GC(v)$ will both be empty. Therefore $W(v) = w_v$ for a leaf from the second case. This means that this particular recurrence doesn't need an explicit base case, because it sort of comes built in to the sum over the children.

Step 4: Analysis This is our first example of a DP where the runtime needs a more sophisticated analysis than just multiplying the number of subproblems by the work per subproblem. If we were to do that naive analysis, we would get $O(n^2)$ since there are $O(n)$ subproblems and

each might have to loop over up to $O(n)$ children/grandchildren. However, we can do better. Note that each vertex is only the child or grandchild of exactly one other vertex (its parent or its grandparent respectively). Therefore, each subproblem is only ever referred to by at most two other vertices. So we can do the analysis "in reverse" or "upside down" in some sense, and argue that each subproblem is only used at most twice, and hence the total work done is just two times the number of subproblems, or $O(n)$.

4 Longest Increasing Subsequence

Our next problem is the "longest increasing subsequence" (LIS) problem, which has an $O(n^2)$ solution, but can then be improved with some clever optimizations!

Problem: Longest Increasing Subsequence

Given a sequence of comparable elements $a_1, a_2, \ldots, a_n$, an increasing subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_{k-1}}, a_{i_k}$ $(i_1 < i_2 < \ldots i_k)$ such that

$$a_{i_1} < a_{i_2} < \ldots < a_{i_{k-1}} < a_{i_k}.$$

A longest increasing subsequence is an increasing subsequence such that no other increasing subsequence is longer.

Find some optimal substructure Given a sequence $a_1, \ldots, a_n$ and its LIS $a_{i_1}, \ldots, a_{i_{k-1}}, a_{i_k}$, what can we say about $a_{i_1}, \ldots, a_{i_k-1}$? Since $a_{i_1}, \ldots, a_{i_k}$ is an LIS, it must be the case that $a_{i_1}, \ldots, a_{i_{k-1}}$ is an LIS of $a_1, \ldots a_{i_k}$ such that $a_{i_{k-1}} < a_{i_k}$. Alternatively, it is also an LIS that ends at (and contains) $a_{i_{k-1}}$. This suggests a set of subproblems.

Define our subproblems Lets define our subproblems to be

$$\text{LIS}[i] = \text{the length of a longest increasing subsequence of } a_1, \ldots a_i \text{ that contains } a_i$$

Note that the answer to the original problem **is not** necessarily LIS$[n]$ since the answer might not contain a_n, so the actual answer is

$$\text{answer} = \max_{1 \le i \le n} \text{LIS}[i]$$

Deriving a recurrence Since LIS$[i]$ ends a subsequence with element i, the previous element must be anything a_j before i such that $a_j < a_i$, so we can try all possibilities and take the best one

$$\text{LIS}[i] = \begin{cases} 0 & \text{if } i = 0, \\ 1 + \max\limits_{\substack{0 \le j < i \\ a_j < a_i}} LIS[j] & \text{otherwise.} \end{cases}$$

Analysis We have $O(n)$ subproblems and each one takes $O(n)$ time to evaluate, so we can evaluate this DP in $O(n^2)$ time. Is this a good solution or can we do better?

4.1 Optimizing the runtime: better data structures

The by-the-definition implementation of the recurrence for LIS gives an $O(n^2)$ algorithm, but sometimes we can speed up DP algorithms by solving the recurrence more cleverly. Specifically in this case, the recurrence is computing a **minimum over a range**, which sounds like something we know how to do faster than $O(n)$...

How about we try to apply a range query data structure (a SegTree) to this problem! Initially, its not clear why this would work, because although we are doing a range query over $1 \le j < i$, we have to account for the the constraint that $a_j < a_i$, so we can not simply do a range query over the values of LIS$[1 \ldots (i-1)]$ or this might include larger elements.

So here's an idea... Let's solve the subproblems *in order* by the value of the final element, instead of just left-to-right by order of i. That way, when we solve a particular subproblem corresponding to a particular final element, we will have only processed the subproblems corresponding to all smaller elements, which are all legal to append the next larger element to!

```
function LIS(a : list⟨int⟩) -> int = {
  sortedByVal := sorted list of (value, index) pairs
  // SegTree is endowed with the RangeMax operation
  results := SegTree(array⟨int⟩(size(a), 0))
  for val, index in sortedByVal do {
    answer := results.RangeMax(0, index) + 1  // Solve the subproblem
    results.Assign(index, answer)             // Store the subproblem
  }
  return results.RangeMax(0, size(a))
}
```

This optimized algorithm performs two SegTree operations per iteration, and it has to sort the input, so in total it takes $O(n \log n)$ time.

Key Idea: Speed up DP with data structures

If your DP recurrence involves computing a minimum, or a sum, or searching for something specific, you can sometimes speed it up by storing the results in a data structure other than a plain array (e.g., a SegTree or BST).

Exercises: Dynamic Programming

Problem 1. We showed how to find the weight of the max-weight independent set. Show how to find the actual independent set as well, in $O(n)$ time.

Problem 2. Give an example where using the greedy strategy for the 0-1 knapsack problem will get you less than 1% of the optimal value.

Problem 3. Suppose you are given a tree T with vertex-weights w_v and also an integer $K \geq 1$. You want to find, over all independent sets of cardinality K, the one with maximum weight. (Or say "there exists no independent set of cardinality K".) Give a dynamic programming solution to this problem.

Problem 4. Can you find a greedy algorithm that matches the $O(n \log n)$ performance of the LIS algorithm above?

Problem 5. Write a different implementation of the LIS algorithm that still uses a SegTree but loops over i and uses range queries on j instead (the opposite of the solution above).