Dynamic Programming

COMP 251, Fall 2024

Will Zahary Henderson and Giulia Alberini

This set of notes provides an overview of the dynamic programming paradigm, motivated by examples. These notes roughly match the lectures on November 12 and 14.

Contents

1 What is dynamic programming?

So far in this class, we have studied two classes of algorithms: **greedy** algorithms which follow the heuristic of making the locally optimal choice at each stage to eventually lead to a globally optimal solution, and **divide-and-conquer** algorithms which recursively break down a problem into smaller subproblems until they can be solved directly and then combined into the solution to the original problem.

A problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems. A problem is said to have **overlapping subproblems** if subproblems are reused several times to solve the broader problem rather than always generating new subproblems.

The **dynamic programming** technique is applicable to problems exhibiting both the properties of optimal substructure and overlapping subproblems. The general idea is that we can take a large problem, break it down into some set of (overlapping) subproblems, and then solve the subproblems from smallest to largest, until we eventually reach the solution to our original problem.

One of the key differences between divide-and-conquer and dynamic programming is that divide-and-conquer algorithms solve problems by combining solutions to *non-overlapping* subproblems, and as such, individual subproblems tend to be unique and not useful to solving other subproblems.

1.1 Memoization vs dynamic programming

Sometimes we will hear people differentiate between "top-down" and "bottom-up" approaches to dynamic programming.

Top-down refers to a (generally) recursive approach where the original problem is called and broken down into subproblems recursively, but each subproblem is cached (often called **memoized** in this context, and often done with a hash table) so that it does not need to be recomputed if it comes up again in another recursive call.

Bottom-up refers to a non-recursive approach where the smallest subproblems are computed and stored, allowing subsequent, larger subproblems to be computed and stored, and so on until the original problem is eventually able to be computed; all of the storing is done in some n-dimensional table, usually implemented with an n-dimensional array (where n is usually 1 or 2 in the problems we look at, but could be much larger for more complex problems).

Whether the top-down approach is truly considered dynamic programming is a topic of controversy, but in this class we will refer to it as **memoization** instead. Generally, a (bottom-up) dynamic programming approach is faster in practice when all subproblems need to be calculated as it reduces the overhead of making recursive calls, but memoization may be preferred when only a small subset of the subproblems are needed.

2 Algorithms

2.1 Fibonacci sequence

The Fibonacci sequence $F = (F_0, F_1, \ldots)$ is defined by the recurrence relation

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

We can calculate F_n recursively by simply converting this to code as follows.

`F(n):`

```
1  if n ≤ 1 then
2    return n
3  else
4    return F(n-1) + F(n-2)
```

Here, the number of recursive calls to F grows exponentially, and the algorithm runs in time $O(2^n)$ because we end up having to compute $F(m)$ for $m < n$ multiple times each.

Dynamic programming

We can greatly reduce the time complexity by linearly calculating and storing the values $F(2), F(3), \ldots, F(n-1)$, first, instead of having to recompute them.

`F(n):`

```
1  A ← array of length n+1 (index 0 to n)
2  A[0] ← 0
3  A[1] ← 1
4  for i ← 2 to n do
5    A[i] ← A[i-1] + A[i-2]
6  return A[n]
```

It is easy to see that this runs in time $\Theta(n)$ — much faster!

In fact, since we are only ever looking at the two most recently computed Fibonacci numbers to compute the next one, we can write an algorithm that does the same thing using only $O(1)$ space. I will leave this as an exercise.

2.2 Weighted interval scheduling

When we discussed greedy algorithms, we focused on the **interval scheduling problem**. In this problem, we were provided a set of n activities $S = \{ a_1, a_2, \ldots, a_n \}$ where each activity a_i had a start time s_i and a finish time f_i, and we needed to maximize the number of non-overlapping activities we could do. We were able to solve this simply by sorting the activities by their finish times and greedily selecting the compatible activity with the earliest finish time at each step.

We had also briefly introduced the **weighted interval scheduling problem**, where additionally each activity is assigned a *weight*, and where the objective is to choose the set of activities that maximizes the sum of the weights. Now, we will discuss this problem in more depth and how we can solve it with dynamic programming.

Context

Given a set of n activities $S = \{ a_1, a_2, \ldots, a_n \}$, where each activity a_i has a start time s_i, a finish time f_i, and a weight w_i, we wish to output the maximum weight subset of mutually compatible activities.

In other words, we want to

$$\text{maximize} \sum_{i \in A} w_i$$

subject to $f_j \leq s_k$ or $f_k \leq s_j$ for all $a_j, a_k \in A$ (i.e., a_j and a_k do not overlap).

We will break this problem down into subproblems by keeping track of two new attributes:

- p, where $p[j]$ is the largest index $i < j$ such that activity i is compatible with j, and
- OPT, where OPT$[j]$ is the value of the optimal solution to the problem including activities 1 to j, i.e., the maximum total weight of compatible activities considering only activities 1 to j.

Recursive approach

We can compute OPT recursively, noting that OPT$[n]$ returns the solution to the problem.

We have a binary choice to make at each activity a_i: we can either include a_i in the solution, or not include it.

If we include a_i,

- the weight w_i is used to compute OPT$[i]$,
- we cannot use any activities that are incompatible with a_i,
- we can build the optimal solution by including a_i and then finding the optimal solution among activities a_j for $j \in \{ 1, 2, \ldots, p[i] \}$, and

- the weight of the optimal solution, OPT[i], is equal to w_i + OPT[$p[i]$].

If we do not include a_i,

- we can build the optimal solution by finding the optimal solution among activities a_j for $j \in \{1, 2, \ldots, i-1\}$, and
- the weight of the optimal solution OPT[i] is equal to OPT[$i-1$].

Thus, we have the following recursive breakdown:

$$\text{OPT}[i] = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ w_i + \text{OPT}[p[i]],\ \text{OPT}[i-1] \right\} & \text{otherwise.} \end{cases}$$

Given the number of activities n along with arrays containing their start times (s), finish times (f), and weights (w), we can then calculate the maximum sum of weights recursively. First, we must sort the activities by their finish times and compute p for each activity.

Here, `c-opt` is short for "compute OPT."

`c-opt(i):`

```
1  if i = 0 then
2    return 0
3  else
4    return max(w[i] + c-opt(p[i]), c-opt(i-1))
```

Memoization

Notice that in the recursive algorithm, we may end up calling `c-opt(i)` multiple times for the same i. This is inefficient; we can instead store the results of calls to `opt` so that they can be reused instead of recomputed. This is called **memoization**.

In this case, we can initialize an array with $n + 1$ elements (from 0 to n) called `OPT`. Initially, we set `OPT[0]` to 0 and can set all other entries to -1. We can then adjust our earlier algorithm.

`c-opt(i):`

```
1  if OPT[i] = -1 then
2    OPT[i] ← max(w[i] + c-opt(p[i]), c-opt(i-1))
3  return OPT[i]
```

This memoized version is *much* more efficient!

We analyze the complexity as follows:

- Sorting the activities by finish time takes $O(n \log n)$, as does computing p for each activity.

- Each call to `c-opt(i)` takes time $O(1)$ and either returns an existing value from OPT[i] or fills in the entry by making two recursive calls. Since a call to `c-opt(i)` will only make a recursive call once (the first time it is called), at most $2n$ recursive calls are made in total (two for each n activities).

Thus, the initial call to `c-opt(n)` takes time $O(n)$. The overall algorithm takes time $O(n \log n)$ if we include the time taken to sort the activities and calculate p.

Dynamic programming approach

Instead of memoization, which takes a recursive, top-down approach to solving this problem, we can calculate the results of all the subproblems we need and "build up" to the solution; this is classic *bottom-up* dynamic programming.

Notice that in each call to `c-opt(i)` in the recursive solution, we make recursive calls to `c-opt(i-1)` and to `c-opt(p[i])`. Thus, we can always calculate OPT[i] quickly if we already know OPT[$i-1$] and OPT[$p[i]$].

More generally, we know that if we compute OPT[j] for all $j < i$, we have solved the subproblems we need to compute OPT[i].

We can then present the following version of the algorithm that calculates OPT[n] without making any recursive calls, again assuming we have already calculated p and sorted the activities by finish time.

`dp-opt():`

```
1  OPT[0] ← 0
2  for i ← 1 to n do
3    OPT[i] ← max(w[i] + OPT[p[i]], OPT[i-1])
4  return OPT[n]
```

It is easy to see that this approach also takes time $O(n)$ after the p values have been calculated and the activities have been sorted. We are simply taking a maximum in $O(1)$ for all values from 1 to n.

Like before, this solution provides the maximum sum of weights of compatible activities. If we wish to actually find which activities led to this solution, we can run the following algorithm on the values we computed.

`activity-set(i):`

```
1  if i = 0 then
2    return ∅
3  if w[i] + OPT[p[i]] > OPT[i-1] then
4    return {j} ∪ activity-set(p[i])
5  else
6    return activity-set(i-1)
```

Since there are at most n recursive calls, this takes time $O(n)$.

2.3 The knapsack problem

We are given a set of n items where each item has both a weight $w_i > 0$ and a value $v_i > 0$. We are also given a maximum weight W describing the size of our knapsack.

Goal: find the subset $A \subseteq \{ 1, 2, \ldots, n \}$ of items of *maximum total value* such that the sum of their sizes is at most W (i.e., all of the items fit into the knapsack).

In other words, if $x_i \in \{ 0, 1 \}$ is defined as

$$x_i = \begin{cases} 1 & \text{if we include item } i \text{ in the knapsack} \\ 0 & \text{if we do } \textbf{not} \text{ include } i \text{ in the knapsack,} \end{cases}$$

we want to

$$\text{maximize } \sum_{i=1}^{n} v_i x_i \quad \text{subject to } \sum_{i=1}^{n} w_i x_i \leq W.$$

This is called the **0-1 knapsack problem**, named for the restriction of $x_i \in \{ 0, 1 \}$, and is the most common form of the knapsack problem. It is what we will focus on.

Other forms of the knapsack problem include the **bounded knapsack problem** where the restriction on x_i is loosened to $x_i \in \{ 0, 1, 2, \ldots, k \}$ for some k (i.e., we can reuse items up to k times) and the **unbounded knapsack problem** where x_i is unrestricted beyond being in $\mathbb{N}$.

Naïve approach

For each i, $x_i = 0$ or $x_i = 1$. We can perform a complete search of all 2^n possible configurations by considering all binary strings of length n $\{ x_1, x_2, \ldots, x_n : x_i \in \{ 0, 1 \} \}$. For each of these possibilities, we check that the items do not cross the weight threshold of the knapsack by seeing if $\sum_{i=1}^{n} w_i x_i \leq W$, and if so, we calculate the sum of the values of each i such that $x_i = 1$ and return the largest such sum.

Of course, this strategy is horribly inefficient: we are checking 2^n possible configurations! This takes time $O(2^n)$. We can do better.

Dynamic programming approach

We can approach the 0-1 knapsack problem using dynamic programming. Define $m[i, U]$ to be the maximum sum of values achieved among the first $i \leq n$ items $(1, \ldots, i)$ with weight less than or equal to $U \leq W$.

We can define $m[i, U]$ recursively as follows:

- $m[0, U] = 0$;
- if $w_i > U$, then $m[i, U] = m[i-1, U]$ since we clearly cannot include item i;
- if $w_i \leq U$, then $m[i, U] = \max \{ m[i-1, U], m[i-1, U - w_i] + v_i \}$.

In the last case, $m[i-1, U-w_i]+v_i$ represents the possibility that we wish to add item i to the knapsack, but we must make room for it by checking the maximum value of the knapsack with weight $U-w_i$ so that after adding i, the weight is at most U.

The solution to the knapsack problem is then found by finding $m[n, W]$: the maximum sum of values achieved between all n items with weight less than or equal to W. Since this is calculated recursively, we have to calculate all the $m[i, U]$ for $U \leq W$ and $i \leq n$ first. We can put their values in a "table" or matrix represented as a 2D array.

We can write the dynamic programming algorithm as follows, where we are given n and W as input, along with arrays w and v of length n where $w[i]$ and $v[i]$ represent the weight and value of item i, respectively.

`knapsack(n,W,w,v):`

```
1  m ← (n+1) × (W+1) 0-indexed 2D array
2  for j ← 0 to W do
3    m[0,j] ← 0
4  for i ← 0 to n do
5    m[i,0] ← 0
6
7  for i ← 1 to n do
8    for j ← 1 to W do
9      if w[i] > j then
10       m[i,j] ← m[i-1,j]
11     else
12       m[i,j] ← max(m[i-1,j], m[i-1,j-w[i]] + v[i])
13
14 return m[n,W]
```

Note that this only solves the problem of finding the maximum sum of values but does not actually return the set of items that leads to this maximum. If we wish to return that set, we can work backwards from $m[n, W]$ and check where we added new items, noting that if $m[i, U] > m[i-1, U]$, then we must have added item i to the knapsack.

We perform this search recursively on our existing 2D array `m` as follows, also assuming we still have access to the `w` and `v` arrays.

`knapsack-set(i,j):`

```
1  if i = 0 then
2    return ∅
3  if m[i,j] > m[i-1,j] then
4    return {i} ∪ knapsack-set(i-1,j-w[i])
5  else
6    return knapsack-set(i-1,j)
```

We can then return our desired set by calling `knapsack-set(n,W)`.

Complexity

Of course, it is not hard to see that `knapsack` takes time $\Theta(nW)$ since this is the number of cells in the table we have to compute, and computing one cell takes time $O(1)$. `knapsack-set` is a simple $O(n)$ additional pass, so it does not add any complexity if we run it alongside the original `knapsack` algorithm.

Example

As a short example to illustrate the algorithm at work, consider a knapsack with maximum weight $W = 6$ and the following $n = 5$ items:

i	1	2	3	4	5
w_i	2	1	4	3	6
v_i	4	3	1	5	7

Following the algorithm, we obtain the following m table.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4
2	0	3	4	7	7	7	7
3	0	3	4	7	7	7	8
4	0	3	4	7	8	9	12
5	0	3	4	7	8	9	12

We see that $m[n, W] = m[5, 6] = 12$, so the maximum sum of values in the knapsack is 12.

To find the items that lead to this optimal solution, we call `knapsack-set(5,6)` and can visualize the path the matrix follows:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4
2	0	3	4	7	7	7	7
3	0	3	4	7	7	7	8
4	0	3	4	7	8	9	12
5	0	3	4	7	8	9	12

where gray cells represent cells where $m[i, j] \leq m[i-1, j]$ and blue cells represent cells where $m[i, j] > m[i-1, j]$ that we include in the solution. Thus, the solution set is items $\{1, 2, 4\}$.

3 Pseudo-polynomial time

Note: we are not going to test you on this, but it is good to know.

Let's talk about what $O(nW)$ means in the knapsack problem — believe it or not, we do *not* consider this to be polynomial time. It *does* run in time polynomial with respect to the number of items n and with respect to the *numeric value* of the weight requirement W. However, it does *not* run in a time polynomial with respect to the *length* of the input W, referring to the number of bits used to represent it.

For the most part, the algorithms we have seen in this class have dealt with length of input to mean the number of items in a data structure we provide; for instance, the number of integers in an array, the number of nodes in a tree, or the number of edges in a graph. Perhaps non-intuitively, when we have an *integer* input like W, we define the **length of the integer input** to be the number of bits used to represent it rather than the actual value of the integer.

In the case of the knapsack problem, this means that increasing the length of W means adding an extra bit, which *doubles* the time complexity of the algorithm since we need twice as many columns in the table. If w is the number of bits needed to represent W (i.e., the "length" of W), then the time complexity of the knapsack problem is $O(n2^w)$. Nonetheless, we still typically will write the time complexity as $O(nW)$ — confusing, I know!

In fact, even the simple Fibonacci sequence algorithm from section 2.1 runs in pseudo-polynomial time, for the same reason.

3.1 Complexity classes and consequences

*Note: we are **especially** not going to test you on this; you will learn it in COMP 360. I have decided to include it in case it sparks an early interest. :)*

It is important to make the distinction between polynomial time and pseudo-polynomial time in the field of complexity theory. Pseudo-polynomial time algorithms technically belong to the class of exponential algorithms.

The general class of decision problems that an algorithm can solve in polynomial time (i.e., can be solved "quickly") is referred to as **P**. The general class of decision problems that *cannot* necessarily be solved quickly but for which answers can be *verified* quickly (that is, if given a valid solution to the problem, we can verify that the solution indeed holds in polynomial time) is referred to as **NP**, or **nondeterministic polynomial time**. Every NP problem has an exponential-time algorithm; this is because there are always exponentially-many results that we can verify in polynomial time. Note that $\mathrm{P} \subseteq \mathrm{NP}$.

The decision problem version of the knapsack problem (which asks whether we can attain at least value V with maximum weight W) belongs to NP, and in fact is **NP-complete**, meaning that it is among the set of the hardest problems in NP.

The **P versus NP problem** is one of the most well-known unsolved problems in theoretical computer science, though it is widely theorized that $P \neq NP$. If $P = NP$, then we could solve any NP problem (including the knapsack problem) in polynomial time. Since the knapsack problem is NP-complete, if we could find a polynomial time algorithm to solve it, this would be enough to prove that $P = NP$!

For further reading, I recommend the Wikipedia page on NP-completeness. I also highly recommend taking COMP 360 if you find this sort of thing interesting.

4 Exercises

Exercise 1. The **fractional knapsack problem** is a version of the knapsack problem where we can take any fraction $x_i \in [0, 1]$ of an item i rather than having to either select it ($x_i = 1$) or not select it ($x_i = 0$). Write an algorithm to solve the fractional knapsack problem efficiently. Do we need to use DP, or is there a simpler way? Is the complexity still pseudo-polynomial time, $O(nW)$, or can we do better?

Exercise 2. In the notes, we saw a solution to the 0–1 knapsack problem, but we also defined other versions of the problem: the *bounded* knapsack problem and the *unbounded* knapsack problem. Write dynamic programming algorithms that solve these. Is the time complexity still $O(nW)$?

Exercise 3. The **subset sum problem**[1] is similar to the knapsack problem, and is actually a special case of it. It is an NP-complete **decision problem**, meaning that it outputs a yes or no answer. The problem is: given a multiset[2] S of integers and a target sum T, can a subset of S sum precisely to T? Solve this problem in pseudo-polynomial time using dynamic programming.

[1] https://en.wikipedia.org/wiki/Subset_sum_problem

[2] A **multiset** is a set that can contain repeated values. For example, $\{1, 3, 4, 1, 2\}$ is a multiset and contains two copies of 1.

References

[1] Anna Brandenberger. Dynamic programming. https://luc.devroye.org/AnnaBrandenberger-DynamicProgramming-McGillUniversity-2019.pdf, 2018.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.