14 Knapsack

- Dynamic Programming
- Knapsack
- Sequence Alignment

In general, a dynamic programming algorithm works by choosing a set of subproblems and relating them through some recurrence.
To be efficient, we need:

- Only polynomially many subproblems
- The solution of a problem should be efficiently computable from the solutions of its subproblems
- We can express the solution in terms of a reurrence involving only "smaller" subproblems

14.1 Knapsack

We are given n items, where item i has a positive integer weight w_i (for all $i \in \{1, \cdots, n\}$).

We are given an upper bound W on the total weight we can fit into our knapsack.

Our goal is to find a subset $S \subseteq \{1, \cdots, n\}$ that maximizes the total weight $\sum_{i \in S} w_i$ subject to the constraint that $\sum_{i \in S} w_i \leq W$.

To give a dynamic programming algorithm, what should the subproblems be?

The natural approach would be to just consider items $\{1, \cdots, j\}$.
Let $\text{opt}(j)$ = optimal solution on items $\{1, \cdots, j\}$.
Starting from the last item: is item n part of the optimal solution?

- If not, then $\text{opt}(n) = \text{opt}(n-1)$.
- If yes, then $\text{opt}(n) = w_n + \text{opt}(n-1)$ with a lower weight limit $W - w_n$.

Our subproblems are not rich enough to express this idea of a different weight limit!

Instead, let $\text{opt}(j, w)$ be the optimal solutions on items $\{1, \cdots, j\}$ with weight limit w.

Our goal is to compute $\text{opt}(n, W)$.

We ask if item j is part of the optimal solution on items $\{1, \cdots, j\}$ with total weight $\leq$ w?

- If not, then $\text{opt}(j, w) = \text{opt}(j-1, w)$
- If yes, then $\text{opt}(j, w) = w_j + \text{opt}(j-1, w - w_j)$.

We set the base case here to be $\text{opt}(0, w) = 0$ for any w.

Recurrence:

$$\text{opt(j, w)} = \begin{cases} \text{opt}(j-1, w) & \text{if } w_j > w \\ \max\{\text{opt}(j-1, w), w_j + \text{opt}(j-1, w - w_j)\} & \text{if } w_j \leq w \end{cases}$$

```
Knapsack(n, W):
    let M be an (n+1)x(W+1) array of integers
    let M[0, w] = 0 for all w from 0 to W
    for j=1 to n
        for w=0 to W
            if w_j > w then
                let M[j, w] = M[j-1, w]
            else
                let M[j, w] = max{M[j-1, w], w_j + M[j-1, w-w_j]}
```

```
            endif
        endfor
    endfor
    return M[n, W]
```

Running time: $O(n \cdot W)$ since the body of the loop takes time $O(1)$.

This algorithm is linear in n, but exponential in $\log W$.