Dynamic Programming 1 (Alpha)

Sorting with Dynamic Programming

General problem: Given *n* items, choose some of them (where the order in which you choose them matters) to optimize some value.

- The fact that order matters makes this problem more challenging than straightforward knapsack

Sorting with Dynamic Programming

Recall: The knapsack problem gives N items and a knapsack size of W, with the k'th item having weight $w[k]$ and value $v[k]$. The goal is to maximize the total value of items you select while keeping total weight below W.

This can be solved with dynamic programming using a DP of the form:

$dp(k, w)$ = maximum value that can be achieved using the first *k* objects with a knapsack of size *w*

$$= \max(\text{don't take item } k, \quad \text{take item } k)$$

$$= \max(dp(k-1, w), \quad dp(k-1, w - w[k]) + v[k])$$

Important: This approach works because the prefix of the first *k* objects is guaranteed to be "optimal" in the sense that choosing objects in a different order doesn't matter – all that matters is which objects you choose.

Sorting with Dynamic Programming

Approach: If order matters, first sort the items in a way that is “optimal.” Whichever subset of items you choose, now they will already be in the optimal order amongst themself.

This means that once sorted, we can apply a standard knapsack approach. The challenge is often finding the appropriate sort comparator…

Tip: Consider the final “optimal” order and think about what happens if you swap two items in that order

Sorting with Dynamic Programming

Example problem: (Energy Stones - Google Kickstart Round B 2019)

Duda the rock monster has collected **N** *energy stones* for lunch. He eats energy stones one at a time. The i'th stone takes him $\mathbf{S_i}$ seconds to eat.

Different stones give him different amounts of energy. Furthermore, the stones lose energy over time.
The i'th stone initially contains $\mathbf{E_i}$ units of energy and will lose $\mathbf{L_i}$ units of energy each second.

When Duda starts to eat a stone, he will receive all the energy the stone contains immediately (no matter how much time it takes to actually finish eating the stone). The stone's energy stops decreasing once it hits zero.
What is the largest amount of energy Duda could receive from eating his stones?

- $1 \leq N \leq 100$
- $1 \leq S_i \leq 100$
- $1 \leq E_i \leq 10^5$
- $0 \leq L_i \leq 10^5$

Sorting with Dynamic Programming

Observations with a small case

Stones	1	2	3
Initial energy	10	20	30
Decay rate (per second)	1	0	2
Time to consume	2	1	1

- Stone 2 doesn't decay at all, so eat it last
- If we eat stone 3 before stone 1, then overall loss of energy is 1 * 1 (time to eat stone 3 * decay rate of stone 1)
- If we eat stone 1 before stone 3, then overall loss of energy is 2 * 2 (time to eat stone 1 * decay rate of stone 3)

Therefore, eat stone 3 first then stone 1 (to minimize overall loss of energy) and finally stone 2.

Suggests possible sorting order? (Stones which minimize loss of energy consumed first)

Sorting with Dynamic Programming

Example problem: (Energy Stones - Google Kickstart Round B 2019)

Consider an order of eating stones for Duda. Assume that every stone in this order gives Duda some non-zero amount of energy (it doesn't matter when Duda eats the fully decayed stones since they give 0 energy anyway).

This ordering looks like a permutation of the numbers $\{1,2 \dots, x\}$: $P_1 P_2 P_3 \dots P_x$

Consider two adjacent stones $P_i P_{i+1}$…

- If P_i is eaten first, then we lose $S_i L_{i+1}$ energy overall from P_{i+1}
- If we swap the stones so P_{i+1} is eaten first, we lose *up to* $S_{i+1} L_i$ energy overall (*up to* because it is possible that P_i will decay completely in the time it takes us to eat P_{i+1})

The stone which *minimizes overall energy loss* should be eaten first.

If $S_{i+1} L_i < S_i L_{i+1}$ then eating stone P_{i+1} first reduces overall loss and the optimal ordering is $P_{i+1} P_i$. Otherwise, $P_i P_{i+1}$ is the optimal ordering.

Sorting with Dynamic Programming

Example problem: (Energy Stones - Google Kickstart Round B 2019)

Our comparator: $S_x L_y < S_x L_y$ for any two stones x and y
(Equivalent to sorting the stones from least to most overall energy loss)

Sorting the stones like this guarantees that we can consume any subset and they will already be in an optimal order within that subset. We just need to decide which stones we choose to eat or don't choose to eat…

This is just standard knapsack!

(Implementation left as an exercise)

State Swapping in DP

A simple example:

There are N items, numbered 1,2,…,N. For each *i* ($1 \leq i \leq N$), Item *i* has a weight of w_i and a value of v_i. Choose some of the N items and put them in a knapsack. The capacity of the knapsack is W, which means that the sum of the weights of items taken must be at most W.

Find the maximum possible sum of the values of items that you can choose.

$1 \leq N \leq 100$
$1 \leq W \leq 10^9$ (**notice this constraint** – normal knapsack doesn't work!)
$1 \leq w_i \leq W$
$1 \leq v_i \leq 10^3$

State Swapping in DP

A simple example:

Usual knapsack approach:

$dp(i, w) =$ best score that can be achieved with the first *i* items and a knapsack of size *w*

This is no longer feasible - can't calculate this value over all *w* because *w* goes up to 10^9

- Instead, let's consider swapping the state (knapsack size) and the value (best score that can be achieved)…

State Swapping in DP

A simple example:

Let $dp(i, v) =$ the minimum-size knapsack that is required to achieve an overall value of v using the first *i* items

Our transitions are then:

$$dp(i, v) = \min \begin{cases} dp(i-1, v - v[i]) + w[i] & (take\ i'th\ item) \\ dp(i-1, v) & (leave\ i'th\ item) \end{cases}$$

This is feasible because *v* only goes up to 10^5 (each item has value up to 10^3 and there are up to 100 items).

(More) State Swapping in DP

Common problem: Find the length of the longest increasing subsequence (LIS) of an array A of *N* numbers

Example: LIS of A = {1, 3, 2, 5, 1, 7, 9, 3} has length 5 coming from {**1**, 3, **2**, **5**, 1, **7**, **9**, 3}
Note that this solution is not unique: {**1**, **3**, 2, **5**, 1, **7**, **9**, 3}

(More) State Swapping in DP

O(N^2) DP Solution to LIS

- Let $dp(i)$ = length of longest increasing subsequence ending at position *i* in the array

Then, for each element in the array, we calculate:

$$dp(i) = \max_{1 \le k < i,\ A[i] > A[k]} dp(k) + 1$$

Intuitively: For each element to the left of the *i'th* element in the array, if A[i] is larger than previous element, then take the previous element's longest subsequence and extend it by 1 using the *i'th* element:

A:	**1**	**3**	**2**	**5**	**1**	**7**	**9**	**3**
dp(i):	1	2	2	3	1	4	5	3

(More) State Swapping in DP

O(N^2) DP Solution to LIS

For each element, we check every element before it, so the number of operations is

$$\sum_{i=1}^{N}\sum_{k=1}^{i-1} 1 = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(N^2)$$

We can do better than this… let's swap the state with the value in the DP:

Currently, our DP solution maintains the "best" increasing subsequence at each position *i*.
If we swap these, we maintain the best position for each possible increasing subsequence…

(More) State Swapping in DP

O($NlogN$) DP Solution to LIS

Let dp(i, k) = the *smallest* element out of the first *i* elements in the array which *ends* an increasing subsequence of length k:

A[i]	1	3	2	5	1	7	8	3
dp(i, 1)	**1**	1	1	1	1	1	1	1
dp(i, 2)	-	**3**	**2**	2	2	2	2	2
dp(i, 3)	-	-	-	**5**	5	5	5	3
dp(i, 4)	-	-	-	-	-	**7**	7	7
dp(i, 5)	-	-	-	-	-	-	**8**	8
dp(i, 6)	-	-	-	-	-	-	-	-

It should be clear that as you go down a column in this table (increasing *k*), the values are increasing because as the subsequence gets longer the “end” of an increasing subsequence must increase.

(More) State Swapping in DP

O($NlogN$) DP Solution to LIS

A[i]	1	3	2	5	1	7	8	3
dp(i, 1)	**1**	1	1	1	1	1	1	1
dp(i, 2)	-	**3**	**2**	2	2	2	2	2
dp(i, 3)	-	-	-	**5**	5	5	5	3
dp(i, 4)	-	-	-	-	-	**7**	7	7
dp(i, 5)	-	-	-	-	-	-	**8**	8
dp(i, 6)	-	-	-	-	-	-	-	-

If we go from left to right in the array, we can maintain the corresponding column in the above table as a separate DP array which tracks the *best* subsequence of length *k*.

For an array element A[i], we find the first element that is less than A[i] in the DP array and replace the element immediately after it with A[i].

(More) State Swapping in DP

O($NlogN$) DP Solution to LIS

A[i]	1	3	2	5	1	7	8	3
dp(i, 1)	**1**	1	1	1	1	1	1	1
dp(i, 2)	-	**3**	**2**	2	2	2	2	2
dp(i, 3)	-	-	-	**5**	5	5	5	3
dp(i, 4)	-	-	-	-	-	**7**	7	7
dp(i, 5)	-	-	-	-	-	-	**8**	8
dp(i, 6)	-	-	-	-	-	-	-	-

Example:
Suppose we are at A[2] = 2 in the array.

Initially, we have $dp = \{1, 3, -, -, \dots\}$ (the orange column)
Until this point, the best inc. subsequence of length 1 ends with 1 and the best inc. subsequence of length 2 ends with 3 (and there's nothing longer).

Now find the *largest* element that is *smaller* than A[2] = 2 in the DP array, which is 1. This is the first increasing subsequence we can extend with A[2]. Replace the element immediately after it with A[2].
Now our (updated) dp array at position 2 is $dp = \{1, 2, -, -, \dots\}$

(More) State Swapping in DP

O($NlogN$) DP Solution to LIS

A[i]	1	3	2	5	1	7	8	3
dp(i, 1)	**1**	1	1	1	1	1	1	1
dp(i, 2)	-	**3**	**2**	2	2	2	2	2
dp(i, 3)	-	-	-	**5**	5	5	5	3
dp(i, 4)	-	-	-	-	-	**7**	7	7
dp(i, 5)	-	-	-	-	-	-	**8**	8
dp(i, 6)	-	-	-	-	-	-	-	-

To find the *largest* element in the DP array that is *smaller* than the current array element, we can perform a binary search as the DP array must be strictly increasing (it represents a column in the above table).

Therefore, we iterate over the whole array and at each element perform a binary search on the DP array to find the element to replace. At the end, the length of our LIS is the size of the DP array.

Dynamic Programming with Bitmasks

Recall:

Numbers are represented internally in a computer in binary digits (bits). We can perform operations on these bits directly…

Bit operations

- x & y = AND
- x | y = OR
- x ^ y = XOR
- x << y = left shift (shift all bits left *y* times)
- x >> y = right shift (shift all bits right *y* times)

Dynamic Programming with Bitmasks

Bitmask
We can assign each bit of a number to represent a Boolean. The number representing this set of Booleans is the bitmask.

Example:

Bitmask	Binary representation	Set of Booleans
1	1	{True}
5	101	{True, False, True}
7	111	{True, True, True}
23	10111	{True, False, True, True, True}

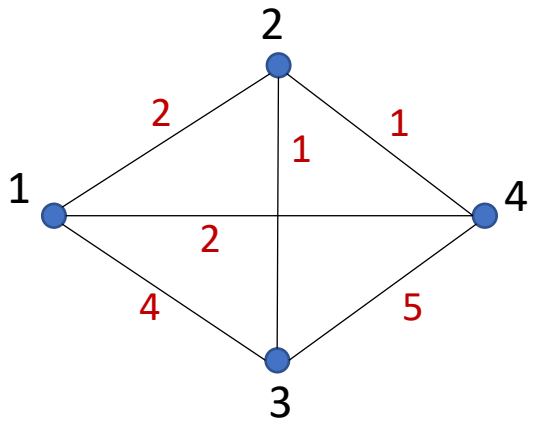
Often used in DP states for representing a set efficiently: If the *k'th* bit of the mask is 1 (from the right), then the *k'th* element is contained within the subset being considered in this state.

Example: Consider the set $S = \{a, b, c\}$
$5 = 101_2 \Leftrightarrow$ {a, c} (keep the first and third elements in the subset)
$6 = 110_2 \Leftrightarrow$ {b, c} (keep the second and third elements in the subset)

Travelling Salesman Problem (TSP) with Bitmask DP

Travelling Salesman Problem: Given a graph which has costs associated with each edge, what is the minimum-cost route that visits every node exactly once and returns to the starting point?

Example:



Possible routes (starting and ending at node 1, ignoring cyclic symmetry):

1 → 2 → 3 → 4 → 1 (cost = 10)

1 → 2 → 4 → 3 → 1 (cost = 12)

1 → 3 → 2 → 4 → 1 (cost = 8) ---------- optimal solution

Travelling Salesman Problem (TSP) with Bitmask DP

Naïve solution:

- Fix node 1 as start and end point
- Try every permutation of remaining (*n*-1) nodes
- Calculate cost of each permutation (and whether edges exist between each pair of consecutive nodes in the permutation*)
- Keep track of min-cost permutation and return this at the end

Runtime: O((n-1)! * n) = O(n!)

** Implementation note: Can represent "lack of an edge" with an edge of cost ∞ and not worry about this*

Travelling Salesman Problem (TSP) with Bitmask DP

Dynamic programming solution:
The naïve solution consider the same path many times
(e.g. cyclic symmetry: 1 → 2 → 3 → 1 is the same path as 1 → 3 → 2 → 1 but will be checked twice as 3,2 and 2,3 are different permutations)

To avoid this, we consider a set of nodes that have already been visited (without considering order):

Let $dp(i, S)$ = min-cost to reach node *i* from node 1, having **already** visited the nodes in set *S.*

Then, we have the following:

$$dp(i,S) = \begin{cases} cost(1,i) & \text{If } S = \{1,2,\dots,N\} \text{ (every node has been visited)} \\ \min_{1 \le x \le N, x \notin S} dp(x, S \cup \{x\}) & \text{Otherwise} \end{cases}$$

Intuitively: $dp(i, S)$ is the min-cost having visited the nodes in *S* and ending at node *i*. From this stage, we need to try visit every node we haven't yet seen (the ones not in S). If we've seen every node, then we need to return to the starting node which requires an additional $cost(1, i)$.

Travelling Salesman Problem (TSP) with Bitmask DP

Dynamic programming solution:

The states in this DP are all subsets of $\{1,2, \dots, N\}$ paired with the last element we are at. This gives a total of $2^n n$ states that need to be considered. At each state, we try to transition to every other node, leading to an additional factor of n.

So, the total runtime is $O(2^n n^2)$.

(Better than factorial but still exponential)

Practice problems

Core:

1) Startup Muster: https://orac2.info/problem/seln18leaders/
2) LCS on Permutations: https://codeforces.com/gym/102951/problem/C
3) King Arthur (only works on ORAC 1 right now): https://orac.amt.edu.au/cgi-bin/train/problem.pl?problemid=16

Additional:

1) Oddjobs: https://orac2.info/problem/moocoddjobs/
2) Energy stones: https://codingcompetitions.withgoogle.com/kickstart/round/0000000000050eda/00000000001198c3
3) Team Building: https://codeforces.com/contest/1316/problem/E

Extension:

1) Elevator problem: https://cses.fi/problemset/task/1653
2) Coding Company: https://cses.fi/problemset/task/1665 (sorting + dp, hard!)

Additional content (if we finish early)

(More) Converting from Permutations to Subsets with Bitmasks

Example: (Well-known elevator problem)
There are N people who want to get to the top of a building which has only one elevator. You know the weight of each person w_i, and the maximum allowed weight in the elevator, x. What is the minimum number of elevator rides?

- $1 \leq N \leq 20$
- $1 \leq x \leq 10^9$
- $1 \leq w_i \leq x$

(More) Converting from Permutations to Subsets with Bitmasks

Example: (Well-known elevator problem)

Similar to TSP, we can use a bitmask DP to avoid considering all possible *orders* of people entering the elevator and check subsets of people instead ($n! \to 2^n$).

Consider the state:

$dp(S) =$ {least number of elevator rides required for people in set S, weight in the *last* elevator ride}

(Each state stores two values)

Then in each state, we consider excluding each person $\mathrm{k} \in S$ one-by-one (so checking $dp(S \backslash \{k\})$) and using the minimal amount of elevator rides required over all these transitions.

We add 1 to the elevator ride count if the weight in the last elevator ride of the transition state is too large to allow person k to join in the same ride, otherwise we increase the size of the last elevator ride.

(Challenge) Open and Close Intervals Trick

Example: (Coding Company - https://cses.fi/problemset/task/1665**)**

There are *N* coders and each of them has a skill level between 0 and 100. Your task is to divide the coders into teams that work together.

Teams work well when the skill levels of the coders are about the same. For this reason, the penalty for creating a team is the skill level difference between the best and the worst coder.

In how many ways can you divide the coders into teams such that the sum of the penalties is at most *x*?

- $1 \leq N \leq 100$
- $0 \leq x \leq 5000$
- $0 \leq t_i \leq 100$

(Challenge) Open and Close Intervals Trick

Example: (Coding Company - https://cses.fi/problemset/task/1665**)**

Solution outline - the DP state is:

$dp(i, j, k) =$ number of ways with first *i* (sorted) elements, *j* "open" groups, and with total penalty *k*. An "open" group is one that is not yet finished.

The transitions are then:

- $dp(i, j, k) \mathrel{+}= dp(i-1, j, k)$ (Add person *i* to their own group)
- $dp(i, j, k) \mathrel{+}= dp(i-1, j-1, k + skill[i])$ (Create new group with just person *i*)
- $dp(i, j, k) \mathrel{+}= j * dp(i-1, j, k)$ (Add person *i* to unfinished group – don't finish it)
- $dp(i, j, k) \mathrel{+}= (j+1) * dp(i-1, j+1, k - skill[i])$ (Add person *i* to unfinished group – finish it)