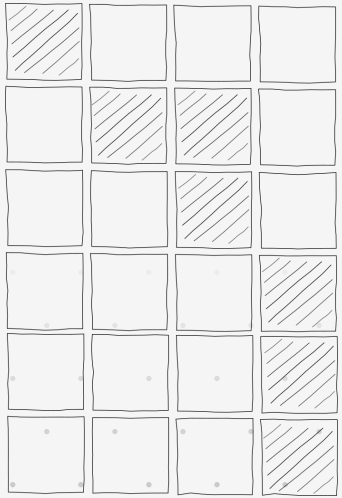
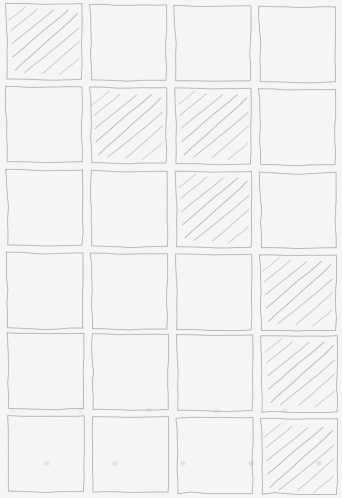


Algorithms ▸ Dynamic Programming

Lecture 2

- Dynamic Programming Review
- Knapsack
- Dynamic Programming on Trees
- Optimizing Dynamic Programs



Algorithms ▸ Dynamic Programming

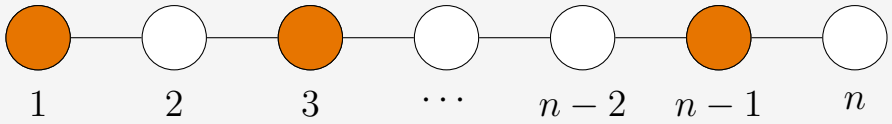
Lecture 2

- Dynamic Programming Review
- Knapsack
- Dynamic Programming on Trees
- Optimizing Dynamic Programs

Orange Nodes Problem

Problem

Positions $1, \ldots, n$, with $n \geq 1$, have nonnegative rewards $w_1, \ldots, w_n$. Choose positions with maximum total reward, with no two chosen positions adjacent.



Orange positions form a feasible selection.

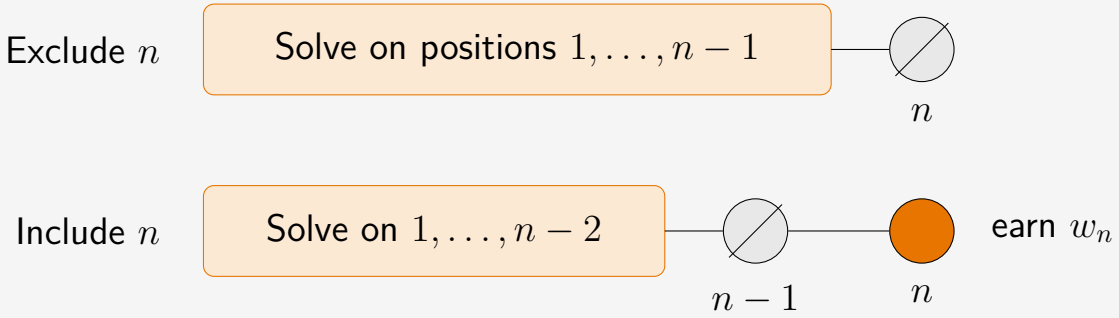
We could try every subset and keep the best feasible one: 2^n possibilities.

Remark

Can we organize the choices so that smaller problems reappear?

The Choice at the Last Position

Every selection either excludes the last position or includes it.



Each case leaves the same selection problem on a shorter prefix. We can solve both cases recursively and keep the better reward.

A Recurrence for Prefixes

The recursive calls ask for the best reward on a prefix. Let $D[i]$ denote that answer for positions $1, \ldots, i$.

$$D[0] = 0, \qquad D[1] = w_1,$$
$$D[i] = \max\{\underbrace{D[i-1]}_{\text{exclude } i}, \quad \underbrace{w_i + D[i-2]}_{\text{include } i}\} \qquad (i \geq 2).$$

Proof

Every feasible selection either excludes i, or includes i and excludes $i-1$. In each case, an optimal selection for the remaining prefix gives a feasible selection of the displayed value. Induction on i proves the recurrence correct.

The answer to the original problem is $D[n]$.

Evaluation Order and Total Work

Each state $D[i]$ depends only on $D[i-1]$ and $D[i-2]$.

- **Top-down:** recurse from $D[n]$ and memoize[1] each state.
- **Bottom-up:** compute $D[0], D[1], \ldots, D[n]$ in order.

There are $O(n)$ states and $O(1)$ work per state, so the running time is $O(n)$.

Store all values in $O(n)$ space.

Note. The bottom-up implementation can use $O(1)$ extra space by keeping only the previous two values.

[1] *Memoizing* means to store the results of recursive calls so that they are computed at most once.

Top-Down and Bottom-Up Pseudocode

Algorithm: Top-down with memoization

Initialize every $M[i]$ to a special `unset` marker.
`Solve`(i):

1. If $i \leq 0$, return 0.
2. If $i = 1$, return w_1.
3. If $M[i] =$ `unset`, set

$$M[i] \leftarrow \max\{\texttt{Solve}(i-1),\ w_i + \texttt{Solve}(i-2)\}.$$

4. Return $M[i]$.

Output `Solve`(n).

Algorithm: Bottom-up tabulation

Set $D[0] \leftarrow 0$ and $D[1] \leftarrow w_1$.
For $i = 2, \ldots, n$:

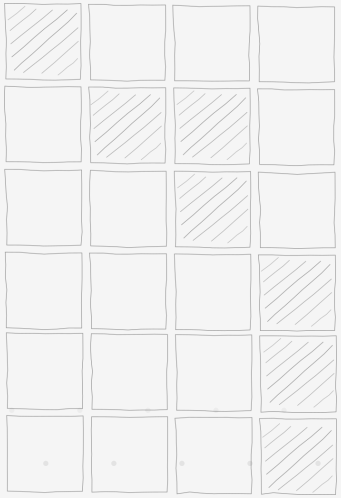
$$D[i] \leftarrow \max\{D[i-1],\ w_i + D[i-2]\}.$$

Output $D[n]$.

Lessons: Dynamic Programming

Organize the choices into subproblems. When the same subproblem reappears, reuse its answer.

1. **Define the states precisely.** Specify what each answer means, the base cases, and the answer to the original problem.
2. **Derive and justify the recurrence.** Cover every possible case and explain why the subproblem solutions combine correctly.
3. **Count the work.** Count states and the work needed for their transitions.
4. **Choose storage and evaluation order.** Memoize recursive calls or fill the table with dependencies first.



Algorithms ▸ Dynamic Programming

Lecture 2

- Dynamic Programming Review
- Knapsack
- Dynamic Programming on Trees
- Optimizing Dynamic Programs

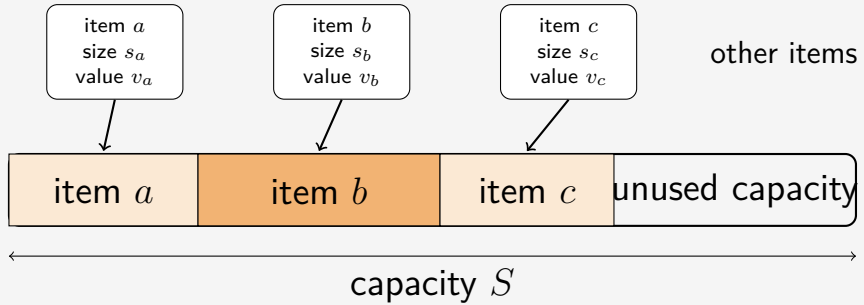
The Knapsack Problem

Problem

The capacity S is a nonnegative integer. Item i has positive integer size s_i and nonnegative integer value v_i. Pick a subset $\mathcal{S} \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in \mathcal{S}} s_i \leq S$$

and $\sum_{i \in \mathcal{S}} v_i$ is maximum. Each item is available once.



Unless $\mathrm{P} = \mathrm{NP}$, this problem has no polynomial-time algorithm.

First Attempt: One State per Prefix

Borrow the prefix state from the path problem:

$$A[i] = \text{maximum value using items } 1, \ldots, i \text{ at capacity } S.$$

- **Exclude i:** the best value is $A[i-1]$.
- **Include i:** add v_i to a solution using the earlier items.

A tempting recurrence is

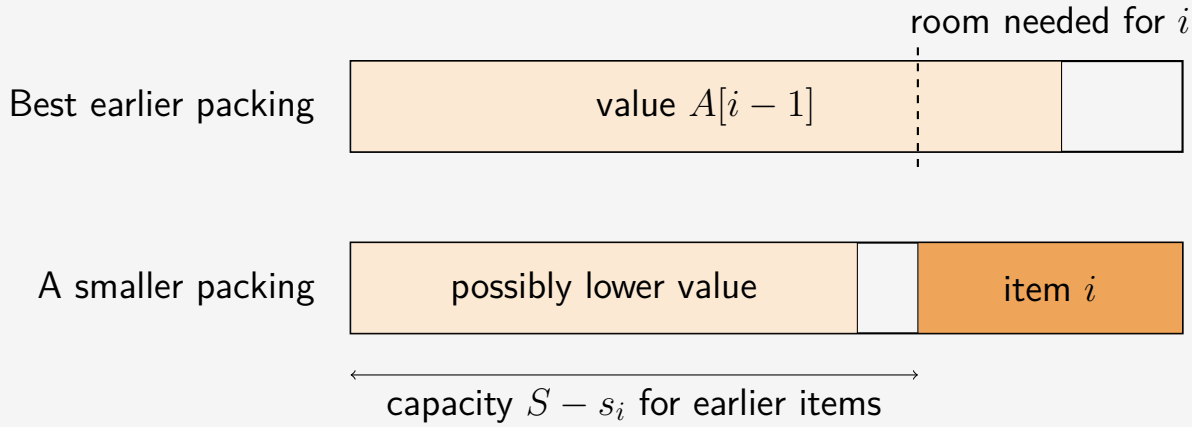
$$A[i] \stackrel{?}{=} \max\{A[i-1],\ v_i + A[i-1]\}.$$

Remark

The state $A[i-1]$ does not encode whether its packing leaves room for item i, so it does not contain enough information to decide the include transition.

Why the One-Dimensional Attempt Fails

The best packing at capacity S may leave too little room for item i.



Adding v_i to $A[i-1]$ can give an infeasible answer. Keeping just that optimum also discards smaller packings that could fit with i.

A Second Attempt: A State for Each Prefix and Capacity

The recursive subproblem depends on both the available items and the capacity.

Define $V[i, B]$ as the maximum value using items $1, \ldots, i$ using at most B units of capacity.

- **Exclude i:** use the first $i - 1$ items at capacity B.
- **Include i:** earn v_i and use the first $i - 1$ items at capacity $B - s_i$, provided $s_i \leq B$.

The old state $A[i]$ is just $V[i, S]$. The second coordinate gives the smaller-capacity answers that were missing.

Knapsack: Recurrence and Correctness

For $B = 0, \ldots, S$, set $V[0, B] = 0$. For $i \geq 1$,

$$V[i, B] = \begin{cases} V[i-1, B], & s_i > B, \\ \max\{V[i-1, B],\ v_i + V[i-1, B - s_i]\}, & s_i \leq B. \end{cases}$$

Proof

Every feasible subset either excludes i or includes it. Including i leaves a subset of the first $i-1$ items with capacity $B - s_i$. Conversely, either subproblem solution gives a feasible subset for its case. Taking the better case and inducting on i proves correctness.

The final answer is $V[n, S]$.

State Dimensions Determine the Cost

Algorithm: Evaluating the knapsack table

Initialize row 0. For $i = 1, \ldots, n$, compute all entries $V[i, 0], \ldots, V[i, S]$ from row $i - 1$. Return $V[n, S]$.

- $(n + 1)(S + 1)$ states, with at most two choices per state.
- $O(nS)$ time and $O(nS)$ space for $n, S \geq 1$.

Note. We can make the amount of space $O(S)$ by keeping only the previous and current rows.

Why isn't this polynomial time?

Capacity $S = 2^b$ takes only $b + 1$ bits to encode in binary. The dynamic-programming table has $S + 1 = 2^b + 1$ columns.

The $O(nS)$ running time depends on the numeric capacity S. It's not polynomial in the number of bits used to describe the input.

The size of the input[2] is $O(nb)$, whereas the running time is $O(n2^b)$.

This is a pseudo-polynomial algorithm: it is polynomial in the numeric value of S, but not necessarily in the input's bit-length.

[2] Assume $s_i, v_i \leq S$ to make things simpler.

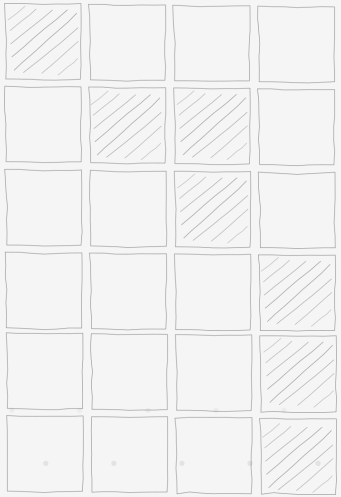
Lessons: Knapsack

Choose state dimensions that fully describe the subproblem and retain all information needed for its transitions.

- The prefix i identifies which items are available.
- The capacity B identifies which subsets fit.
- Together, (i, B) lets us make either choice and describe exactly the subproblem that remains.

Remark

A state may summarize many different solutions, but it must not discard information that could change a later decision.



Algorithms ▸ Dynamic Programming

Lecture 2

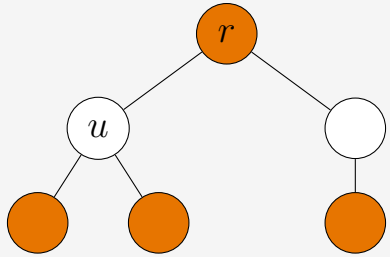
- Dynamic Programming Review
- Knapsack
- Dynamic Programming on Trees
- Optimizing Dynamic Programs

Orange Nodes Problem on a Tree

Problem

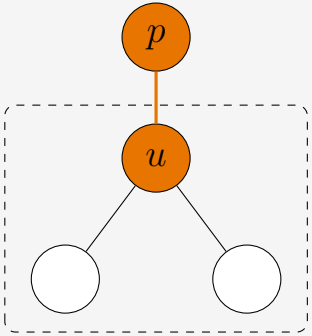
A tree has a nonnegative reward w_v at each vertex. Choose vertices of maximum total reward, with no two chosen vertices adjacent. Such a selection is an independent set.

Root the tree at r. The subtree rooted at v contains v and all its descendants.



Orange vertices show one feasible independent set.

The Parent Needs a Boundary Condition



A subtree optimum might include u.

Selecting p makes that choice incompatible.

We need the best selection in the subtree that excludes u, even when its unrestricted optimum includes u.

This hints that we want to model states as subtrees, and we need two states per subtree.

Two States per Subtree

For the subtree rooted at v, define

$$\mathrm{IN}[v] = \text{maximum reward when } v \text{ is included},$$
$$\mathrm{OUT}[v] = \text{maximum reward when } v \text{ is excluded}.$$

If $C(v)$ is the set of children of v, then

$$\mathrm{IN}[v] = w_v + \sum_{u \in C(v)} \mathrm{OUT}[u],$$

$$\mathrm{OUT}[v] = \sum_{u \in C(v)} \max\{\mathrm{IN}[u], \mathrm{OUT}[u]\}.$$

Different child subtrees have no edges between them. Once we fix v's status, their choices are independent and their rewards add.

Why the Boundary Summary Is Sufficient

Proof

If v is included, every child must be excluded. If v is excluded, either status is allowed for each child.

Every feasible selection restricts to one of these choices in each child subtree. Conversely, the chosen child solutions combine into a feasible selection, because there are no edges between different child subtrees. Induction on subtree size proves both recurrences.

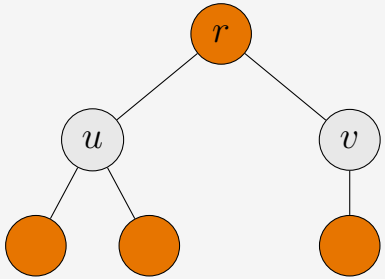
For a leaf v, the empty sums give

$$\mathrm{IN}[v] = w_v, \qquad \mathrm{OUT}[v] = 0.$$

At the root r, the answer is $\max\{\mathrm{IN}[r], \mathrm{OUT}[r]\}$.

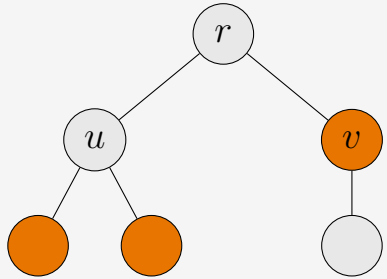
Combining Subtree Solutions

Include the root



Use OUT at every child.

Exclude the root



Choose IN or OUT separately at each child.

Orange vertices show illustrative feasible selections. The rewards determine which allowed selection is best within each child subtree.

Tree DP Pseudocode

Algorithm: Maximum-weight independent set on a rooted tree

`Solve(`v`)`:

1. Set $\mathrm{IN}[v] \leftarrow w_v$ and $\mathrm{OUT}[v] \leftarrow 0$.
2. For every child u of v:
   - Call `Solve(`u`)`.
   - Set $\mathrm{IN}[v] \leftarrow \mathrm{IN}[v] + \mathrm{OUT}[u]$.
   - Set $\mathrm{OUT}[v] \leftarrow \mathrm{OUT}[v] + \max\{\mathrm{IN}[u], \mathrm{OUT}[u]\}$.
3. Return $\max\{\mathrm{IN}[v], \mathrm{OUT}[v]\}$.

Call `Solve(`r`)` at the root.

Analyzing the Algorithm

The algorithm in the previous slide processes vertices in postorder, i.e., every child is processed before its parent.

Computing both states at v takes $O(1 + |C(v)|)$ work. Thus

$$\text{total work} = O\left(\sum_{v}(1 + |C(v)|)\right) = O(n),$$

since each of the $n - 1$ parent–child edges contributes once.
Two values per vertex use $O(n)$ space, including traversal storage.

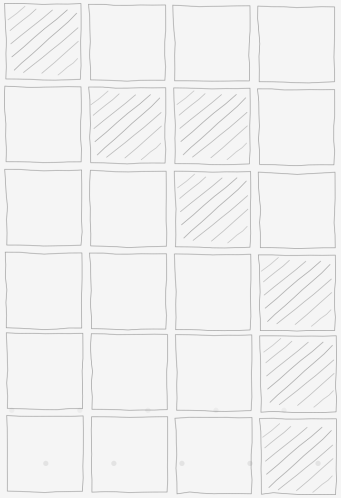
Lessons: Tree DP

The object we do dynamic programming over does not have to be linear. Subproblems can be rooted subtrees instead of prefixes of a sequence.

- The tree tells us how to decompose the problem and order the computation.
- Once we fix a vertex's status, its child subtrees can be solved independently.

Remark

Why does the solution from the previous slides fail on a general graph? If we root a spanning tree, what could the remaining edges change?



Algorithms ▸ Dynamic Programming

Lecture 2

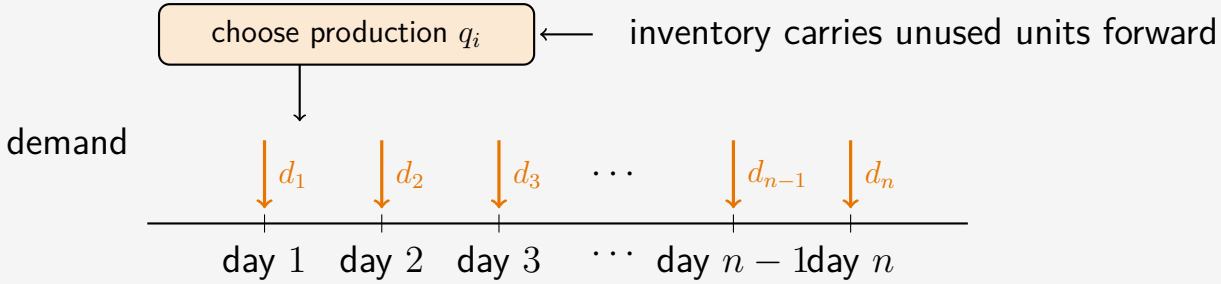
- Dynamic Programming Review
- Knapsack
- Dynamic Programming on Trees
- Optimizing Dynamic Programs

The Production Problem

Problem

A factory must meet positive integer demand d_i on each day i. Producing any positive quantity costs a setup fee $K > 0$. Storing one unit overnight costs $h > 0$. Minimize the total setup and storage cost.

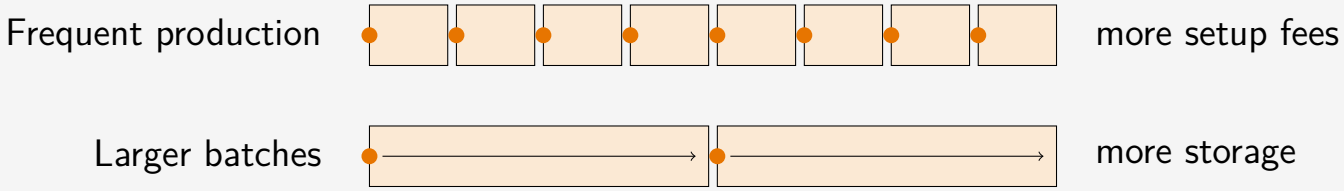
Start and end with no inventory. Meet demand on time. Production and storage are unlimited. Identical per-unit production costs can be omitted.



Choose production quantities so inventory never becomes negative.

Production Tradeoff

Producing frequently reduces storage, but each production day incurs a setup fee. Producing larger batches reduces setup fees, but keeps more units in storage.



Orange dots mark production days. Each bar spans the days supplied by one batch.

First Model: Day and Inventory

Let $A[i, x]$ be the minimum cost through day i, with x units left, including storage charges for that night's inventory.
To reach state (i, x), suppose the preceding state leaves y units after day $i-1$. We must produce the following number of units on day i:

$$q = d_i + x - y \geq 0$$

We pay K when $q > 0$, then pay hx to store the x units remaining overnight. Minimizing over every feasible y gives

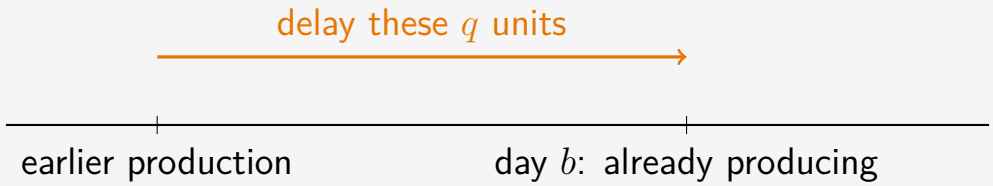
$$A[i, x] = hx + \min_{\substack{0 \leq y \leq D \\ y \leq d_i + x}} \left\{ A[i-1, y] + K \, \mathbf{1}_{\{d_i + x - y > 0\}} \right\},$$

where $D = \sum_i d_i$.

- Initialize $A[0, 0] = 0$ and $A[0, x] = +\infty$ for $x > 0$.
- For each day i and ending inventory $x \in \{0, \ldots, D\}$, scan all feasible preceding inventories y.
- The answer is $A[n, 0]$. There are $O(nD)$ states and $O(D)$ choices per state, for $O(nD^2)$ time.

Producing Later Saves Storage

Suppose we produce on day b while $q > 0$ older units remain just before production. Those units will meet demand on day b or later.



Proof

Remove those units from their earlier batches and produce them on day b. All demand is still met on time. No new setup is needed, and unlimited production permits the extra units. Storage cost strictly decreases.

An optimal plan has zero inventory immediately before each production day. Each batch therefore supplies one consecutive block of days.

Second Model: Batch Boundaries

Let $F[i]$ be the minimum cost to supply days $1, \ldots, i$ and end with no inventory.
If the final batch starts on day j, it supplies exactly days $j, \ldots, i$. Let $C[j, i]$ be the cost of that batch.
Then

$$F[0] = 0, \qquad F[i] = \min_{1 \le j \le i} \{F[j-1] + C[j, i]\}.$$

Proof

The argument from the previous slide guarantees an optimal plan of this form. Its earlier batches solve the prefix ending on day $j - 1$. Conversely, appending the batch for $j, \ldots, i$ to any such prefix plan is feasible.

The final answer is $F[n]$. The state now records only a day.

Precomputing Batch Costs

Producing on day j for days $j, \ldots, i$ costs

$$C[j, i] = K + h \sum_{t=j}^{i} (t - j) d_t.$$

The d_t units for day t spend $t - j$ nights in storage.
We can compute all batch costs with dynamic programming. For a fixed start day j, extending the batch by day i gives

$$C[j, j] = K, \qquad C[j, i] = C[j, i - 1] + h(i - j) d_i \quad (i > j).$$

Algorithm: Precompute C

For each $j = 1, \ldots, n$:

1. Set $C[j, j] \leftarrow K$.
2. For $i = j + 1, \ldots, n$, in increasing order, set

$$C[j, i] \leftarrow C[j, i - 1] + h(i - j) d_i.$$

Final Algorithm Analysis

Precomputation: The table $C[j, i]$ has $\binom{n+1}{2} = O(n^2)$ entries, and the recurrence computes each entry in $O(1)$ time.
Optimization: Starting with $F[0] = 0$, compute $F[1], \ldots, F[n]$ in increasing order using

$$F[i] = \min_{1 \le j \le i} \{F[j-1] + C[j, i]\}$$

Each state $F[i]$ examines i possible starts. Thus optimization also takes

$$\sum_{i=1}^{n} i = O(n^2) \text{ time.}$$

The complete algorithm runs in $O(n^2)$ time and uses $O(n^2)$ space for the precomputed batch-cost table.

Lessons: Choosing the Model

The model determines which states and transitions we need. A different model can make the same problem much faster to solve.

- **Day and inventory:** $O(nD)$ states and $O(D)$ choices per state, giving $O(nD^2)$ time, where $D = \sum_i d_i$.
- **Batch boundaries:** an observation lets us remove the inventory dimension, leaving $O(n)$ optimization states.
- Precomputing batch costs makes each transition constant-time, giving $O(n^2)$ total time.

Remark

After finding a correct DP, ask whether the structure of optimal solutions allows a simpler state or a cheaper transition.