Greedy Algorithms and Dynamic Programming

Making Locally Optimal Choices

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

What is a Greedy Algorithm?

Greedy Algorithm: At each step, make the choice that looks best *right now* and never reconsider it.

Three properties:

- Locally optimal at every step
- Irrevocable: no backtracking
- No look-ahead

Key question: When does a locally optimal choice lead to a globally optimal solution?

Advantages

- Simple and intuitive to design
- Easy to implement
- Often $O(n \log n)$ or better

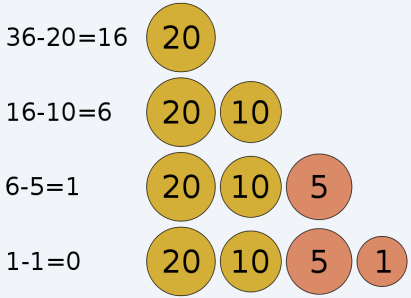
Disadvantages

- Doesn't always give the global optimum
- Correctness requires careful proof
- Hard to know *when* it works

Note Unlike DP, greedy never reconsiders. That's its strength and its weakness.

Problem 1: Coin change

You have coins of various denominations and need to make exact change for an amount T. You want to use as few coins as possible.



Discussion: Can you think of a simple strategy? Will it always give the minimum number of coins?

Problem 1: Coin change

You have coins of various denominations and need to make exact change for an amount T. You want to use as few coins as possible.

Formally: given denominations $d_1 > d_2 > \cdots > d_k$ (with $d_k = 1$), find non-negative integers $x_1, \ldots, x_k$ (number of each coin used) such that

$$\sum_{i=1}^{k} x_i \cdot d_i = T \qquad \text{and} \qquad \sum_{i=1}^{k} x_i \text{ is minimized.}$$

The $d_k = 1$ assumption guarantees a solution always exists.

Discussion: Can you think of a simple strategy? Will it always give the minimum number of coins?

Problem 1: Coin change

Greedy rule: At each step, take the largest coin that fits the remaining amount.

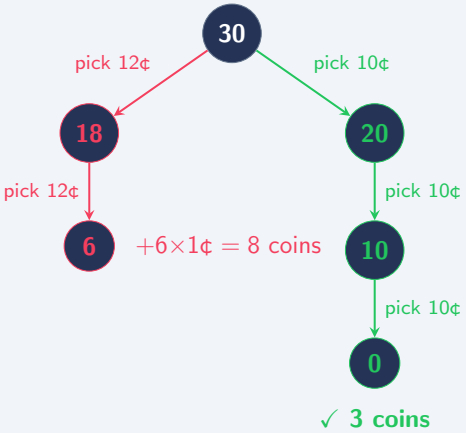
Step 1: take 25¢ (remainder: 5¢)
Step 2: take 5¢ (remainder: 0¢)
✓ **Result: 2 coins — optimal.**

Why it works for US coins: Each denomination divides evenly into larger ones, so taking the biggest coin never blocks a better solution. The greedy structure is baked into the denomination set itself.

But this relies on a special property of US coins. Change the denominations and greedy can fail badly.

Coin change: when greedy fails

Now use denominations $\{12, 10, 1\}$. This but breaks the divisibility property that made US coins safe. Greedy still picks the largest coin first.



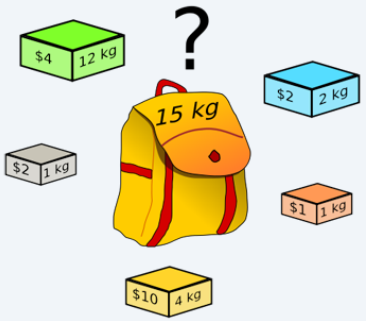
Greedy always picks the largest coin, so it takes 12¢ twice, landing on remainder 6. From there only 1¢ coins fit: **8 coins total.**

The optimal path picks 10¢ three times, reaching remainder 0 in three steps: **3 coins.**

Why greedy fails here: Taking the largest coin first is not a safe greedy choice — there is no exchange argument that makes it work for arbitrary denominations. DP finds the optimum by exploring all branches and remembering the best subproblem solutions.

Problem 2: The knapsack problem

A knapsack can carry at most W kg. A room contains n items, each with a weight w_i and a dollar value v_i. You want to maximize the value you walk out with.

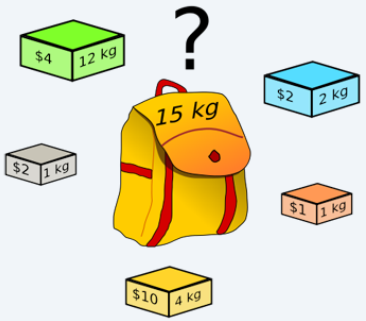


Two variants differ only in what x_i can be:

- **Fractional:** $x_i \in [0, 1]$ — you can scoop out part of an item (gold dust, grain...)
- **0/1:** $x_i \in \{0, 1\}$ — each item is indivisible; take it whole or leave it.

Problem 2: The knapsack problem

A knapsack can carry at most W kg. A room contains n items, each with a weight w_i and a dollar value v_i. You want to maximize the value you walk out with.



Two variants differ only in what x_i can be:

- **Fractional:** $x_i \in [0, 1]$ — you can scoop out part of an item (gold dust, grain...)
- **0/1:** $x_i \in \{0, 1\}$ — each item is indivisible; take it whole or leave it.

Discussion: The two variants look almost identical. Do you think greedy works for one, both, or neither? What strategy would you try?

Problem 2: The knapsack problem

A knapsack can carry at most W kg. A room contains n items, each with a weight w_i and a dollar value v_i. You want to maximize the value you walk out with.

Formally: find quantities x_i to solve

$$\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \qquad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le W, \quad x_i \ge 0.$$

Two variants differ only in what x_i can be:

- **Fractional:** $x_i \in [0, 1]$ — you can scoop out part of an item (gold dust, grain...)
- **0/1:** $x_i \in \{0, 1\}$ — each item is indivisible; take it whole or leave it.

Discussion: The two variants look almost identical. Do you think greedy works for one, both, or neither? What strategy would you try?

Problem 2: Knapsack — two variants

✓ Fractional knapsack — greedy works

You may take any fraction of each item.

Rule: Sort by value/weight ratio; take as much as possible top-down.

Item	Wt	Val	$/kg
Gold dust	1	$60	**$60**
Gems	2	$50	**$25**
Silver bar	4	$80	$20
Bronze	3	$30	$10

Capacity: 5 kg. Take all gold (1 kg), all gems (2 kg), 2 kg of silver. Total: **$160 — optimal.**

Why it works: Any swap of a lower-ratio portion for a higher one would already have been made.

× 0/1 knapsack — greedy fails

Each item: take all or skip — no fractions.

Greedy by ratio takes Gold + Gems (3 kg):

$\$60 + \$50 = \mathbf{\$110}$

But optimal takes Gold + Silver (5 kg):

$\$60 + \$80 = \mathbf{\$140}$

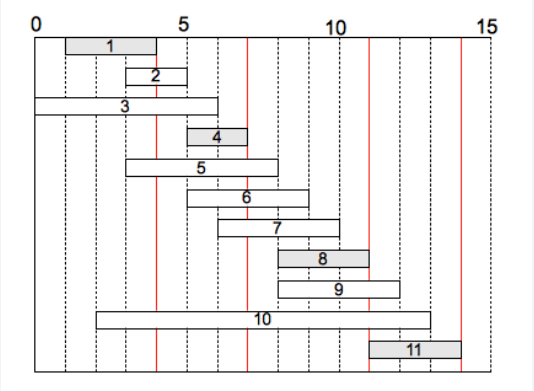
Greedy filled capacity with Gems first and left no room for the more valuable Silver.

Why it fails: Without fractional flexibility, the locally best ratio item can block a better combination. DP explores all 2^n subsets efficiently via tabulation in $O(nW)$.

Problem 3: Activity selection

You manage a single shared resource — a lecture hall, a CPU, a runway. You have n requests, each wanting the resource during an interval $[s_i, f_i)$. Only one activity can use the resource at a time. You want to satisfy as many requests as possible.

Discussion: What rule would you use to pick the next activity to schedule? Try to think of at least two strategies — then ask whether each one actually works.



Problem 3: Activity selection

You manage a single shared resource — a lecture hall, a CPU, a runway. You have n requests, each wanting the resource during an interval $[s_i, f_i)$. Only one activity can use the resource at a time. You want to satisfy as many requests as possible.

Formally: given intervals $[s_i, f_i)$, find the largest subset $S \subseteq \{1, \ldots, n\}$ of pairwise non-overlapping activities:

$$\text{maximize} \quad |S| \qquad \text{subject to} \quad f_i \leq s_j \ \text{ for all } i < j \text{ in } S.$$

Discussion: What rule would you use to pick the next activity to schedule? Try to think of at least two strategies — then ask whether each one actually works.

Problem 3: Activity selection — solution

Three wrong strategies:

× Shortest duration

A short activity can block two long non-overlapping ones.

× Earliest start time

A long early-start activity blocks everything else.

× Fewest conflicts

Too expensive; doesn't generalise.

✓ Earliest finish time

Always pick the compatible activity that *ends soonest*. Leaves maximum remaining time for future picks.

Example timeline teal = selected

Selected: A, D, H, K — 4 activities (optimal)

Complexity: $O(n \log n)$

Why earliest finish time works

Claim: There always exists an optimal solution that includes our greedy choice.

Proof sketch:

1. Let S^* be any optimal solution; let b be its first activity.
2. Let a = our greedy choice (earliest finish).
3. a finishes $\leq b$, so replacing b with a cannot create new conflicts.
4. $S' = (S^* \setminus \{b\}) \cup \{a\}$ is valid and $|S'| = |S^*|$.
5. $\Rightarrow$ **the greedy choice is safe.**

Visual:

Since a finishes $\leq b$, the rest of S^* still fits after the swap. $|S'| = |S^*|$, so S' is also optimal.

After picking a, the remaining problem has the same structure. Apply inductively $\Rightarrow$ optimal.

When does greedy work?

A locally optimal choice at each step leads to a globally optimal solution.

Exchange argument: show any optimal solution can include the greedy choice without becoming worse.

An optimal solution contains optimal solutions to sub-problems. After a greedy choice, the remaining problem has the same structure.

Key distinction: DP needs OS. Greedy also needs GCP. Can't prove GCP? Use DP.

Problem	GCP	OS	Use
Coin change (US)	✓	✓	Greedy
Coin change (arb.)	×	✓	DP
Fractional knapsack	✓	✓	Greedy
0/1 knapsack	×	✓	DP
Activity selection	✓	✓	Greedy
Shortest path (pos.)	✓	✓	Dijkstra's
Shortest path (neg.)	×	✓	Bellman-Ford
MST (Kruskal's)	✓	✓	Greedy

Decision rule:

- Can you prove a safe local choice? → Greedy
- Must you explore all options? → DP
- Not sure? → Try exchange argument. If it fails, use DP.

Greedy vs. dynamic programming

	Greedy	Dynamic Programming
Makes choices	Irrevocably	Considers all options
Subproblems	Independent	Overlapping
Typical complexity	$O(n \log n)$	$O(n^2)$ or higher
Correctness proof	Must prove GCP	Always correct
Optimal substructure	✓	✓
Greedy choice property	✓	×

- Safe local choices exist? → Greedy
- Must explore all options? → DP
- Unsure? → Prove GCP or use DP

- Activity selection → Greedy
- Coin change (arbitrary) → DP
- Dijkstra's (non-negative) → Greedy
- 0/1 knapsack → DP
- MST → Greedy
- Shortest path (negative) → DP

Activity selection: algorithm

```
vector<int> activitySelection(
    vector<pair<int,int>>& acts) {

  // Sort by finish time
  sort(acts.begin(), acts.end(),
    [](auto& a, auto& b){
      return a.second < b.second;
    });

  vector<int> selected;
  selected.push_back(0);
  int lastFinish = acts[0].second;
  for (int i = 1; i < acts.size(); i++) {
    // Compatible: starts >= last finish
    if (acts[i].first >= lastFinish) {
      selected.push_back(i);
      lastFinish = acts[i].second;
    }
  }
  return selected;
}
```

Complexity:

- Sort: $O(n \log n)$
- Loop: $O(n)$
- **Total:** $O(n \log n)$

Space: $O(n)$ for output

- Exchange argument (prev. slide)
- After picking first activity, the rest of the timeline is a sub-problem of identical structure
- Apply inductively

Problem 4 (intro): Minimum spanning tree

Given: Connected, weighted, undirected graph.

Find: Subset of edges connecting all vertices, no cycle, minimum total weight.

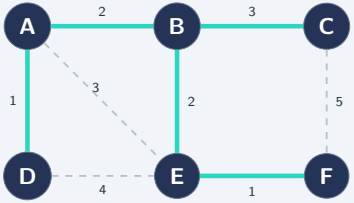
Kruskal's greedy idea:

1. Sort all edges by weight
2. Add each edge if it doesn't create a cycle
3. Stop when $|V| - 1$ edges added

Cut property: the cheapest edge crossing any cut of the graph always belongs to some MST. This is the exchange argument for graphs.

Complexity: $O(E \log E)$ (sort dominates).
Cycle detection: Union-Find, $O(\alpha(V))$ per op.

Full implementation — next lecture.



MST (teal): AD, EF, AB, BE, BC

Total weight: $1 + 1 + 2 + 2 + 3 = 9$

- Cut property $\Rightarrow$ GCP holds
- Sub-graph after each addition has same structure
- OS holds $\Rightarrow$ greedy is correct

Connecting back to graph traversal

These algorithms share a pattern you already know:

BFS:
Explore layer by layer.
Uses a **queue** (FIFO).
Shortest path in unweighted graphs.

Dijkstra's:
Explore cheapest-first.
Uses a **priority queue** (weight order).
Shortest path in weighted graphs.
BFS + greedy ordering.

Kruskal's:
Cheapest edges first.
Uses **Union-Find** for cycle detection (reachability query).
Greedy on edge space.

The unifying idea Greedy algorithms on graphs *are* graph traversals with a smarter ordering. The data structure that enforces that ordering (queue, priority queue, sorted edge list) is what distinguishes them. Union-Find is BFS/DFS reachability made fast with path compression.

Dynamic programming: motivation

We've now seen greedy fail on coin change $\{12, 10, 1\}$ and 0/1 knapsack. The problem in both cases is the same: the locally best choice forecloses a globally better option, and there's no way to go back.

The root cause is *overlapping subproblems* that greedy solves once and discards. DP solves them once and *remembers*.

Two key properties:

Optimal substructure: an optimal solution to the full problem contains optimal solutions to subproblems. (Greedy needs this too.)

Overlapping subproblems: the same subproblems recur many times. Solving each once and caching the result is the whole idea.

Two implementations:

- **Memoization** (top-down): recurse naturally, cache results as you go.
- **Tabulation** (bottom-up): fill a table from smallest subproblems up.

DP: Coin Change Revisited

Denominations $\{12, 10, 1\}$, target 30¢

Recurrence: let $\mathrm{dp}[t]$ = minimum coins to make amount t.

$$\mathrm{dp}[0] = 0 \qquad \mathrm{dp}[t] = \min_{d_i \le t} \big(\mathrm{dp}[t - d_i]\big) + 1$$

Fill the table left to right. Each cell considers all coins that fit:

t	0	1	2	…	10	11	12	13	…	20	21	22	…	28	29	30
dp	0	1	2	…	**1**	2	**1**	2	…	**2**	3	**2**	…	8	9	**3**

$\mathrm{dp}[30] = \mathbf{3}$: three 10¢ coins. Greedy got **8**.

DP found this because it explored *every* subproblem, not just the one greedy's local rule pointed to. The optimal solution to $t = 30$ reused the optimal solution to $t = 20$, which reused $t = 10$: overlapping subproblems, solved once each.

DP: 0/1 knapsack

Items: Gold 1kg/\$60, Gems 2kg/\$50, Silver 4kg/\$80, Bronze 3kg/\$30. Capacity $W = 5$.

Recurrence: let $\mathrm{dp}[i][w]$ = max value using items $1 \dots i$ with capacity w.

$$\mathrm{dp}[i][w] = \max\big(\,\mathrm{dp}[i-1][w],\;\; v_i + \mathrm{dp}[i-1][w-w_i]\big) \quad \text{if } w_i \le w$$

Either skip item i (take $\mathrm{dp}[i-1][w]$) or include it (add v_i, use w_i capacity).

Item	$w=0$	$w=1$	$w=2$	$w=3$	$w=4$	$w=5$
(none)	0	0	0	0	0	0
Gold (1kg, \$60)	0	60	60	60	60	60
+ Gems (2kg, \$50)	0	60	60	110	110	110
+ Silver (4kg, \$80)	0	60	60	110	110	**140**
+ Bronze (3kg, \$30)	0	60	60	110	110	**140**

$\mathrm{dp}[4][5] =$ **\$140** (Gold + Silver). Greedy got **\$110** (Gold + Gems). **Complexity:** $O(nW)$ time and space: pseudopolynomial (polynomial in W, but W can be exponential in the number of input bits).

DP: longest common subsequence (LCS)

A classic problem with no greedy solution.

Problem: given two strings $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$, find the length of their longest common subsequence (characters in order, not necessarily contiguous).

Example: $X =$ `ABCBDAB`, $Y =$ `BDCAB`
$\Rightarrow$ LCS $=$ `BCAB`, length 4.

Recurrence: let $\text{dp}[i][j]$ = LCS length of $X[1..i]$ and $Y[1..j]$.

$$\text{dp}[i][j] = \begin{cases} \text{dp}[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(\text{dp}[i-1][j],\ \text{dp}[i][j-1]) & \text{otherwise} \end{cases}$$

	∅	B	D	C	A	B
∅	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
B	0	1	1	2	2	3
D	0	1	2	2	2	3
A	0	1	2	2	3	3
B	0	1	2	2	3	**4**

DP: edit distance

How many single-character edits to transform one string into another?

	∅	S	I	T	T	I	N	G
∅	0	1	2	3	4	5	6	7
K	1	1	2	3	4	5	6	7
I	2	2	1	2	3	4	5	6
T	3	3	2	1	2	3	4	5
T	4	4	3	2	1	2	3	4
E	5	5	4	3	2	2	3	4
N	6	6	5	4	3	3	2	3

Operations: insert, delete, substitute each costs 1.

$$\text{dp}[i][j] = \begin{cases} \text{dp}[i-1][j-1] & x_i = y_j \\ 1 + \min \begin{cases} \text{dp}[i-1][j] \\ \text{dp}[i][j-1] \\ \text{dp}[i-1][j-1] \end{cases} & \text{o/w} \end{cases}$$

`KITTEN` $\rightarrow$ `SITTING`: distance $= 3$.

Complexity: $O(mn)$ time and space.

Applications: spell checkers, DNA alignment, plagiarism detection, `git diff`.

Greedy vs. DP: the full picture

	Greedy	Dynamic Programming
Strategy	Pick locally best, move on	Solve all subproblems, combine
Backtracks?	Never	No (but considers all options)
Subproblems	Independent after each choice	Overlapping, reused
Typical complexity	$O(n \log n)$	$O(n^2)$ or $O(nW)$
Always correct?	Only with proof of GCP	Yes, given correct recurrence
Optimal substructure	Required	Required
Greedy choice property	Required	Not required

Decision rule:

Can you identify a single locally safe choice at every step and prove it with an exchange argument? $\rightarrow$ **Greedy.**

Summary

Greedy

Locally optimal choice at each step; never reconsider. Correct when GCP + optimal substructure hold.

- Coin change (US coins): $O(n)$
- Fractional knapsack: $O(n \log n)$
- Activity selection: $O(n \log n)$
- MST / Kruskal's: $O(E \log E)$
- Dijkstra's: $O((V + E) \log V)$

Dynamic programming

Solve every subproblem once, store results. Correct whenever optimal substructure holds.

- Coin change (arbitrary): $O(kT)$
- 0/1 knapsack: $O(nW)$
- LCS: $O(mn)$
- Edit distance: $O(mn)$

Key ideas to take away:

Both techniques require *optimal substructure*. Only greedy additionally requires the *greedy choice property*. When GCP holds, greedy is faster. When it doesn't, DP is the fallback.

The same problem, different constraint: fractional knapsack yields to greedy; 0/1 knapsack requires DP. One word in the problem statement changes the algorithm.

DP tables have a shape. 1D tables (coin change) arise from 1D state. 2D tables (knapsack, LCS, edit distance) arise from two independent dimensions. Recognizing the state space is half the work.

Leads To: Complexity theory: P vs. NP, and why some problems resist both greedy and DP entirely.