Algorithm Design and Analysis

Dynamic Programming

Roadmap for today

- Learn about (maybe review) ***dynamic programming***
- Understand the key elements:
  - Memoization
  - Optimal Substructure
  - Overlapping subproblems
- Practice a lot of DP problems!

Starter example: Counting steps

You can climb up the stairs in increments of 1 or 2 steps.
How many ways are there to jump up n stairs?

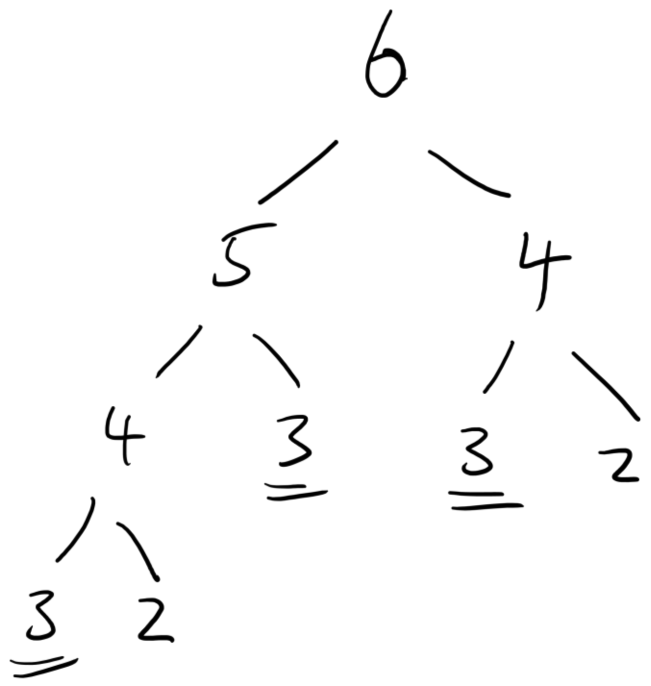
Could we solve this problem in terms of **smaller subproblems**?

$n-1$

$n-2$

Implementation #1

```
function stairs(int n) {
  if (n <= 1) then return 1
  else {
    let waysToTake1Step = stairs(n-1)
    let waysToTake2Steps = stairs(n-2)
    return waysToTake1Step + waysToTake2Steps
  }
}
```



Issue? Exponentially many recursive calls!!

Implementation #2

```
dict<int, int> memo

function stairs(int n) {
  if (n <= 1) then return 1
  if (n not in memo) {
     memo[n] = stairs(n-1) + stairs(n-2)
  }
  return memo[n]
}
```

Key Idea: Memoization

Don't solve the same problem twice! Store the result and reuse it!

When can we use DP?

- We could solve the stairs problem by using solutions to ***smaller*** instances of the stairs problem

```
stairs(n) = stairs(n-1) + stairs(n-2)
```

Key Idea: Optimal substructure

We say that a problem has *optimal substructure* if the optimal solution to the problem can be derived from optimal solutions to smaller instances (called ***subproblems***) of the problem.

When can we use DP?

- The DP implementation of stairs was faster because each subproblem was solved *only once* instead of *exponentially many times*

```
stairs(n) = stairs(n-1) + stairs(n-2)
```

Key Idea: Overlapping subproblems

Overlapping subproblems are subproblems that occur multiple (often exponentially many) times throughout the recursion tree. This is what distinguishes DP from ordinary recursion.

"Recipe" for dynamic programming

1. ***Identify a set of optimal subproblems***
   - Write down a clear and unambiguous definition of the subproblems.
2. ***Identify the relationship between the subproblems***
   - Write down a recurrence that gives the solution to a problem in terms of its subproblems
3. ***Analyze the required runtime***
   - *Usually* (but not always) the number of subproblems multiplied by the time taken to solve a subproblem.

Often all that is required for a theoretical solution

---

4. ***Select a data structure to store subproblems***
   - *Usually* just an array. Occasionally something more complex

Only required if the answer is not "array"

5. ***Choose between bottom-up or top-down implementation***
6. ***Write the code!***

Mostly ignored in this class (unless it's a programming HW!)

The Knapsack Problem

The Knapsack Problem

Definition (Knapsack): Given a set of $\boldsymbol{n}$ **items**, the i^{th} of which has **size** $\boldsymbol{s_i}$ and **value** $\boldsymbol{v_i}$. The goal is to find a subset of the items whose **total size is at most** $\boldsymbol{S}$, with **maximum possible value**.

	A	B	C	D	E	F	G
Value	7	9	5	12	15	6	12
Size	3	4	2	6	7	3	5

$\boldsymbol{S = 15}$

Identifying Optimal Substructure

	A	B	C	D	E	F	G
Value	7	9	5	12	15	6	12
Size	3	4	2	6	7	3	5

↑

Issue:

- How do we know whether to include a particular object X?
- We don't know in advance, so ***try both choices*** and pick best one!

Optimal substructure:

- Every object is either included or not included
- If an item X is included, the remaining S – Size(X) space is filled with some subset of the remaining items
- This is just a smaller instance of the knapsack problem!!

$V(k, B) =$ value of best subset of $\{1 \ldots k\}$ with total size $\leq B$

Writing a recurrence

$$V(k, B) = \begin{cases} 0 & k = 0 \\ V(k-1, B) & \text{if } s_k > B \\ \max\left(V(k-1, B),\ v_k + V(k-1, B - s_k)\right) & \end{cases}$$

Key Idea: Clever brute force

We could not know in advance whether to include the i^{th} item or not, so we tried both possibilities and took the best one.

Analyzing the Runtime

Analysis: Knapsack can be solved in $O(nS)$ time

- $O(nS)$ subproblems
- $O(1)$ per subproblem
- Total: $O(nS)$

Max-weight independent set in a tree (Tree DP)

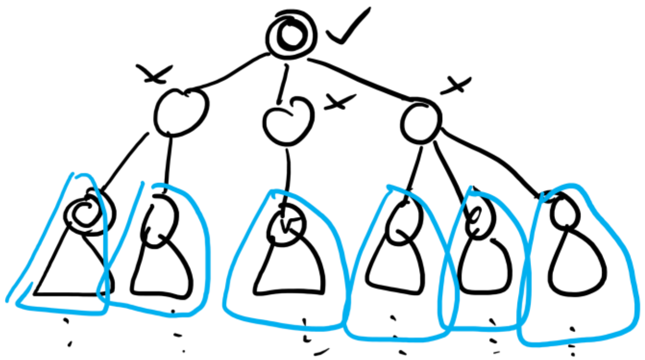
Independent sets on trees (Tree DP)

Definition (Independent set): Given a tree on $\boldsymbol{n}$ **vertices**, an *independent set* is a subset of the vertices $S \subseteq V$ such that none of them are adjacent.

Each vertex has **a non-negative weight $\boldsymbol{w_v}$**, and we want to find the **maximum possible weight** independent set.

Optimal substructure:

- A solution either includes the root or does not include the root
- If the root is chosen, the remaining solution is an independent set of the remaining vertices, excluding the root's children
- Each child/grandchild subtree is just another smaller instance of the MWIS-in-a-tree problem!!



$W(v)$ = value of MWIS of the subtree rooted at v.

Writing a Recurrence

$$W(v) = \max \begin{cases} \sum_{u \in Child(v)} W(u) & (\text{don't choose } v) \\ w_v + \sum_{u \in Grand(v)} W(v) & (\text{choose } v) \end{cases}$$

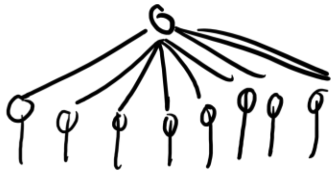
Again: Clever brute force

We could not know in advance whether to include the root or not, so we tried both possibilities and took the best one.

Analyzing the Runtime

Analysis: MWIS on a tree can be solved in $O(n)$!!

$$\text{Total work} = \sum_{v \in V} \#\text{children} + \#\text{grandchildren}$$



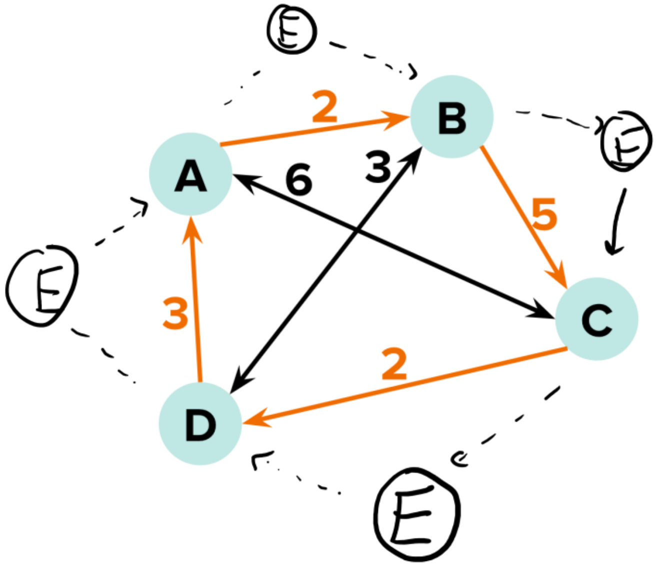
$$= O(n)$$

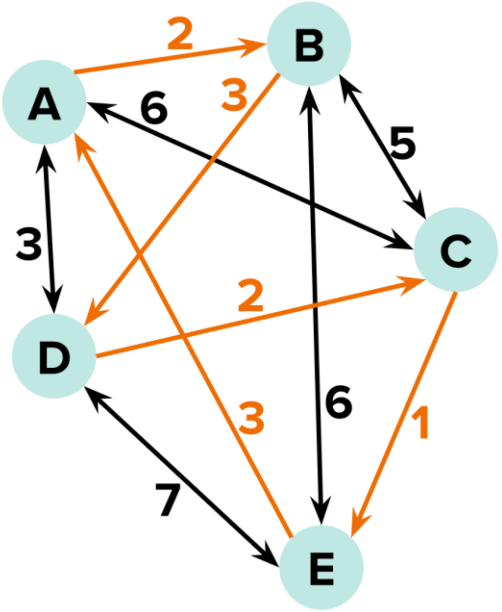
Traveling Salesperson Problem (TSP)

Traveling Salesperson Problem (TSP)

Definition (TSP): Given a complete, directed, weighted graph, we want to find a minimum-weight cycle that visits every vertex exactly once (called a “Hamiltonian Cycle”).

Idea 1: Find the minimum weight cycle on a subgraph with one of the vertices removed, then add that vertex somewhere in the cycle.





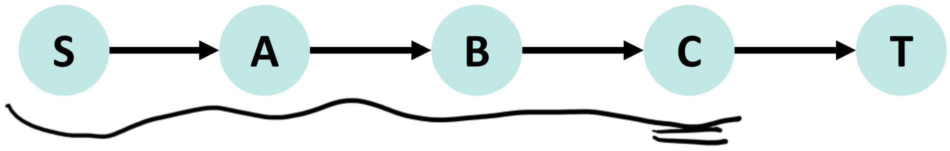
Issue: No obvious optimal substructure. The optimal cycle for {A,B,C,D,E} looks very different to the optimal cycle for {A,B,C,D}

Refining the Subproblems

The issue: Cycles don’t have any obvious optimal substructure

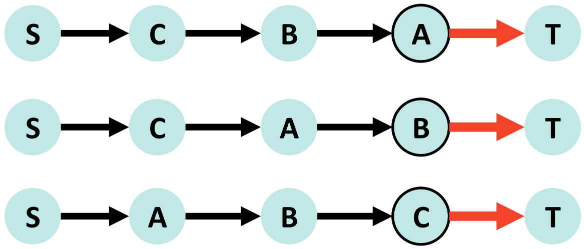
Can we look for another graph property that does?

Paths!



Observe: If $S \to A \to B \to C \to T$ is a minimum weight $S \to T$ path, then $S \to A \to B \to C$ must be a minimum weight $S \to C$ path.

How do we know which vertex to put second last (before T)?



Clever brute force to the rescue!
Try them all and take the best one.

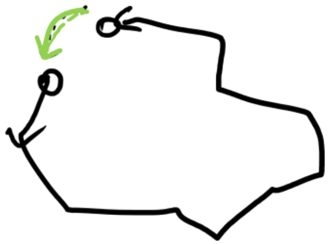
Defining Subproblems

- How should we define subproblems for minimum-weight paths?

$C(S, t)$ = Min-weight path from start to t touching every vertex in subset S

- How do we solve the original problem (TSP) using these subproblems?

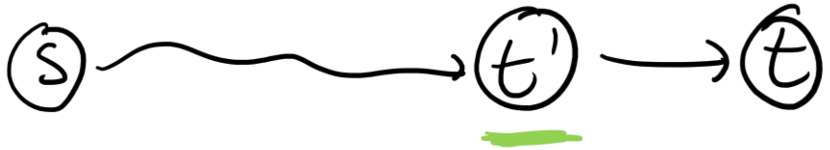
$$TSP = \min_{t' \in V - \{start\}} \left(C(V, t') + w(t', start) \right)$$



Writing a recurrence

- Now we just need the recurrence for minimum weight paths

$$C(S,t) = \begin{cases} w(\text{start}, t) & \text{if } S = \{\text{start}, t\} \\ \min\limits_{\substack{t' \in S \\ t' \notin \{\text{start}, t\}}} C(S - \{t\}, t') + w(t', t) & \end{cases}$$



Analyzing Runtime

Runtime of naïve solution: $O(n!)$

DP solution:

2^n subsets

n final vertices

$O(n)$ work per subproblem

$\rightarrow O(2^n \cdot n^2)$

Take-home messages

- Breaking a problem into subproblems is hard. ***Common patterns:***
  - Can I use the first k elements of the input?
  - Can I restrict an integer parameter (e.g., knapsack size) to a smaller value?
  - On trees, can I solve the problem for each subtree? (Tree DP)
  - Can I solve the problem for a subset of the input (TSP)
  - Can I keep track of more information (start and end vertex in TSP)
- Try a "***clever brute force***" approach.
  - Make one decision at a time and recurse, then take the best thing that results.
  - Can think of this as memoized backtracking
- Complexity analysis is *often* just subproblems $\times$ time per subproblem
  - But sometimes its harder and we must do some more analysis