Algorithm
Design and Analysis

Dynamic Programming (Part II)

Roadmap for today

- More *dynamic programming*

- Review *Longest Increasing Subsequence* (LIS) with SegTrees!

- Derive the *Floyd-Warshall* algorithm for *all-pairs shortest paths*

- See the *Subset DP* technique applied to the *Travelling Salesperson Problem*

"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.

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

5. ***Choose between bottom-up or top-down implementation***

6. ***Write the code!***

Often all that is required for a theoretical solution

Only required if the answer is not "array"

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

Review of LIS

Review of LIS (SegTree DP)

Definition (LIS): Given a sequence of n numbers $a_1, a_2, \ldots, a_n$, find the length of a <u>longest strictly increasing subsequence</u>.

7	0	4	3	10	11	17	15

Subproblems:

$\text{LIS}(i) :=$ The length of the longest increasing subsequence that ends with element a_i (must include a_i)

Recurrence:

$$\text{LIS}(i) = 1 + \max_{\substack{(j \in [0,i)) \\ (a_j < a_i)}} \text{LIS}(j)$$

Optimized LIS: SegTree DP!

$$\text{LIS}(i) = 1 + \max_{\substack{j \in [0, i) \\ a_j < a_i}} \text{LIS}(j)$$

	4	1	3	2	5	6	8	7
A:	7	0	4	3	10	11	17	15

SegTree:	1	1	2	2	3	4	5	5	$LIS(i)$
	LIS(1)	LIS(2)	LIS(3)					LIS(8)	

n SegTree   Range Max + Assign = $O(n \log n)$

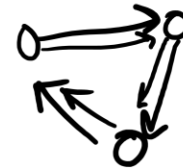# All-pairs shortest paths

All-pairs shortest paths: Attempt 1

Definition (APSP) Given a directed, weighted graph, compute the length of the shortest path between <u>every pair</u> of vertices.

Optimal substructure:
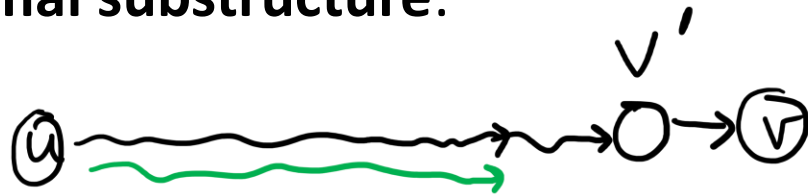
$$SP(u,v) = \min_{v' \in V} SP(u,v') + w(u,v)$$

Subproblems: $SP(u,v,\ell) = $ min-weight u-v path containing at most ℓ vertices

Writing a Recurrence: Attempt 1

$$SP(u, v, \ell) = \begin{cases} \begin{array}{ll} 0 & \text{if } u = v \\ \infty & \text{if } u \neq v \end{array} & \text{if } \ell = 1 \\ \min_{v' \in V} SP(u, v', \ell-1) + w(v', v) \end{cases}$$

Solution: $\min_{\ell} SP(u, v, \ell)$

Analyzing Runtime: Attempt 1

$$\mathrm{SP}(u, v, \ell) = \min_{v' \in V}(\mathrm{SP}(u, v', \ell - 1) + w(v', v))$$

Naïve analysis:

n^3 subproblems

$O(n)$ each

$\Rightarrow O(n^4)$

Better analysis:

$$n \sum_u \sum_v \text{indegree}(v)$$

$$= n^2 \cdot \sum_v \text{indegree}(v)$$

$$= O(n^2 m)$$

All-pairs shortest paths: Attempt 2

> ***Definition*** (APSP) Given a directed, weighted graph, compute the length of the shortest path between <u>every pair</u> of vertices.

Optimal substructure:

Subproblems: $SP(u, v, k) =$ min-weight path from $u \to v$ using only $\{2 \dots k\}$ as intermediate vertices

Writing a Recurrence: Attempt 2

$$\mathrm{SP}(u, v, k) = \begin{cases} \begin{cases} 0 & \text{if } u = v \\ w(u,v) & (u,v) \in E \\ \infty & (u,v) \notin E \end{cases} & \text{if } k = 0 \\ \min\left\{ SP(u,v,k-1), \; SP(u,k,k-1) + SP(k,v,k-1) \right\} \end{cases}$$

Analyzing Runtime: Attempt 2

$$\mathrm{SP}(u, v, k) = \min(\mathrm{SP}(u, v, k-1),$$
$$\mathrm{SP}(u, k, k-1) + \mathrm{SP}(k, v, k-1))$$

$n \quad n \quad n$

Runtime analysis:

n^3 subproblems

$O(1)$ work

$\Rightarrow O(n^3)$ time!

What about space?

$\theta(n^3)$ space

Optimization: Don't store solutions to old values of k. Paths can only stay the same or get shorter as we add more vertices!

Floyd-Warshall Algorithm

```
def floydWarshall(graph G):
  SP[u][v] =    base cases from previous slide


  for k in [1, n]:
    for u in [1, n]:
      for v in [1, n]:
        SP[u][v] = min (SP[u][v],  SP[u][k] + SP[k][v])
  return SP
```

**Exercise:** Prove correctness of the Floyd-Warshall algorithm.

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}
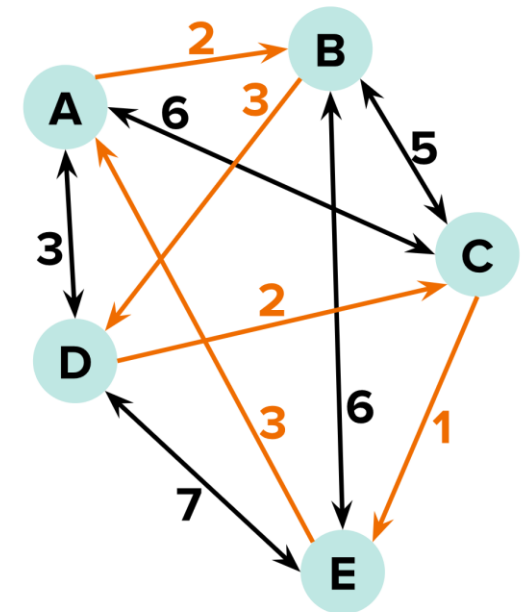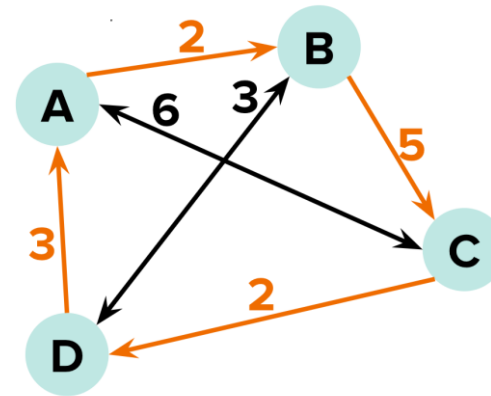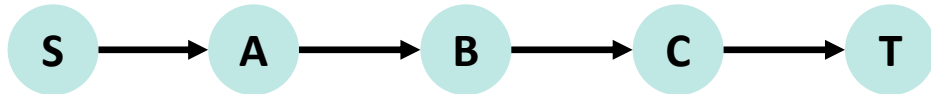
Refining the Subproblems

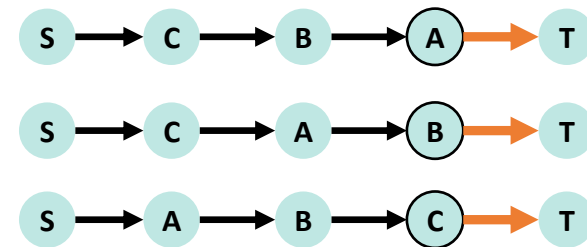**The issue**: Cycles don't have any obvious optimal substructure

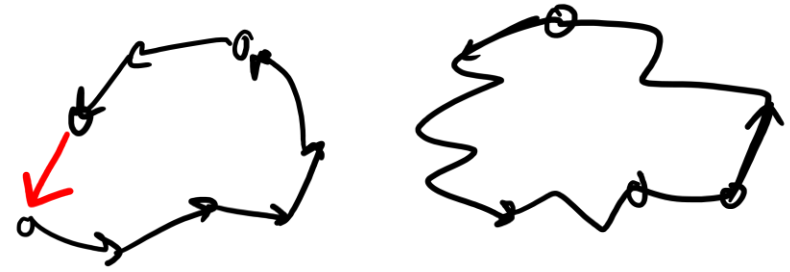Can we look for another graph property that does?

Paths!

Observe: If $S \rightarrow A \rightarrow B \rightarrow C \rightarrow T$ is a minimum weight $S \rightarrow T$ path, then $S \rightarrow A \rightarrow B \rightarrow C$ must be a minimum weight $S \rightarrow C$ 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? _fix_ _✓arbitrarily_

$$\text{MinPath}(S, t) = \text{min-weight path from } \boxed{\text{start}} \text{ to } t, \text{ touching every vertex in } \underline{S}$$

subset of vertices

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

$$\text{TSP} = \min_{t \in V - \{start\}} \left(\text{MinPath}(V, t) + w(t, start) \right)$$

Writing a recurrence
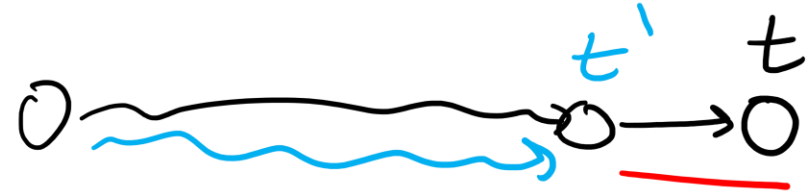
- Now we just need the recurrence for minimum weight paths

$$
\mathrm{MinPath}(S, t) = \begin{cases} w(start, t) & \text{if } S = \{start, t\} \\[2em] \min_{\substack{t' \in S \\ t' \notin \{start, t\}}} \mathrm{MinPath}(S - \{t\}, t') + w(t', t) \end{cases}
$$

Analyzing Runtime

Runtime of naïve solution: $O(n!)$

DP solution: $O(2^n \cdot n)$ subproblems

$O(n)$ work

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

Subset DP: Representing subsets

- Wait, isn't each subset $\Theta(n)$ space and therefore takes $\Theta(n)$ time to look up? So, we actually need more time and space?

Optimization: Represent subsets as **bitsets**. Each subset is represented by a single integer, where the ith bit is 1 if and only if the ith vertex is in the subset.

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 store a subset of the input? (TSP subproblems)
  - Can I remember the most recent decision? (Previous vertex in TSP)

- Many techniques are useful to **optimize** a DP algorithm:
  - Can I remove redundant subproblems to save space? (Floyd-Warshall)
  - Can I use a fancier data structure than an array? (LIS with SegTree)