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 (steps 1–3)

Only required if the answer is not “array” (step 4)

Mostly ignored in this class (unless it’s a programming HW!) (step 5)

Longest Increasing Subsequence

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 longest strictly increasing subsequence.

7	**0**	**4**	**3**	**10**	**11**	**17**	**15**

Subproblems:

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

Recurrence:

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

Optimized LIS: SegTree DP!

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

A:								
	7	**0**	**4**	**3**	**10**	**11**	**17**	**15**

SegTree:								

Optimized LIS: Pseudocode

```
function LIS(list A):
  n = length(A)
  results := SegTree(array of n+1 0’s)
  sortedByVal := sorted list of (val, index) pairs
  for (val, index) in sortedByVal:




  return
```

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 every pair of vertices.

Optimal substructure:

Subproblems:

Writing a Recurrence: Attempt 1

$$\mathrm{SP}(u, v, \ell) = \begin{cases} \\ \\ \end{cases}$$

Analyzing Runtime: Attempt 1

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

Naïve analysis:

Better analysis:

All-pairs shortest paths: Attempt 2

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

Optimal substructure:

Subproblems:

Writing a Recurrence: Attempt 2

$$\mathrm{SP}(u, v, k) = \begin{cases} \\ \\ \end{cases}$$

Analyzing Runtime: Attempt 2

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

Runtime analysis:

What about 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] =

  for k in [1, n]:
    for u in [1, n]:
      for v in [1, n]:
        SP[u][v] =
  return SP
```

Exercise: Prove correctness of the Floyd-Warshall algorithm.

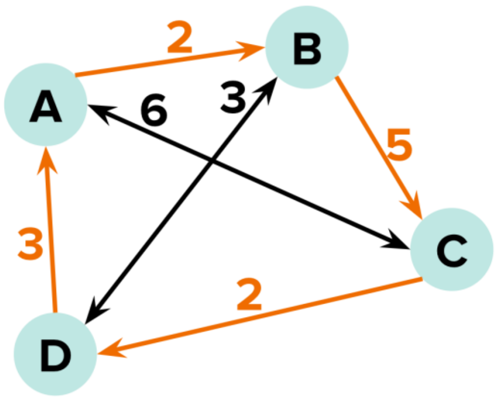
Traveling Salesperson Problem (TSP)

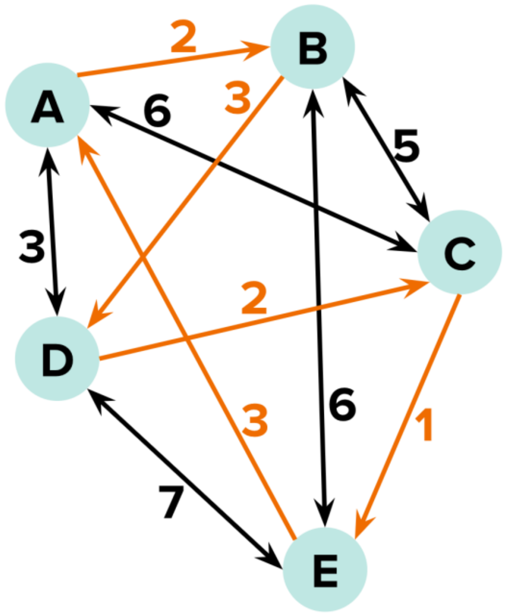
Traveling Salesperson Problem (TSP)

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}



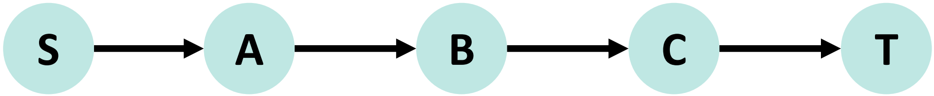


Refining the Subproblems

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

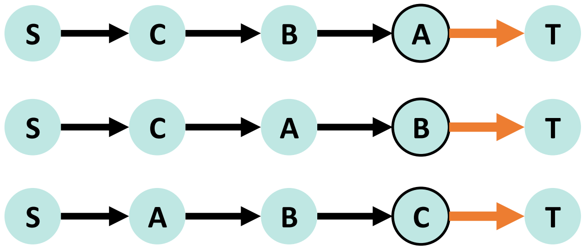
Can we look for another graph property that does?

Paths!



Observe: If $S \to A \to B \to C \to T$ 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?

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

Writing a recurrence

- Now we just need the recurrence for minimum weight paths

$$\text{MinPath}(S, t) = \begin{cases} \\ \\ \end{cases}$$

Analyzing Runtime

Runtime of naïve solution:

DP solution:

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 i^{th} bit is 1 if and only if the i^{th} 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)