Module: Dynamic Programming

Thanks to Anna Lubiw and other previous CS 341 instructors.

Dynamic Programming Basics

The main idea of **dynamic programming** is to solve the subproblems from smaller to larger (bottom up) and store the results as you go.

Dynamic Programming Basics

The main idea of **dynamic programming** is to solve the subproblems from smaller to larger (bottom up) and store the results as you go.

Consider the recursive implementation of Fibonacci:

```
FibR(n)
1.    if $n = 0$ then return 0
2.    elseif $n = 1$ then return 1
3.    else return $FibR(n-1) + FibR(n-2)$
```

Analysis: $T(n) = T(n-1) + T(n-2) + \Theta(1) \in O(2^n)$
$O(2^n)$ recursive calls, combining work is $O(1)$.

Dynamic Programming Basics

A better approach is to use an iterative method and work up from the base cases.

```
FibI(n)
1.    f[0] ← 0
2.    f[1] ← 1
3.    for i ← 2 to n do
4.        f[i] ← f[i − 1] + f[i − 2]
5.    return f[n]
```

Analysis: $O(n)$ arithemtic operations.
Build solutions to $O(n)$ smaller problems (bottom up), each in $O(1)$ time.

Text Segmentation

Problem

Text Segmentation
Instance: *A string of letters $A[1..n]$ where $A[i] \in \{A, \dots, Z\}$.*
Question: *Can A be split into (2 or more) words?*

Assume you are given a tester that runs in $O(1)$ time:

$$Word(i,j) = \begin{cases} \mathit{True} \text{ if } A[i,j] \text{ is a valid word} \\ \mathit{False} \text{ otherwise} \end{cases}$$

Is there a simple solution?

- Find the first (shortest) word that is a prefix ...?
- Find the longest word that is a prefix ...?

Or do you have to check all prefixes between?

Text Segmentation

We can build up a solution for $A[n]$ from smaller subproblems?
Suppose we know Split(k) for $k = 0, 1, \ldots, n-1$ where:

$$Split(k) = \begin{cases} \textit{True} \text{ if } A[1..k] \text{ is splittable} \\ \textit{False} \text{ otherwise} \end{cases}$$

How do we find Split(n)?

Text Segmentation

We can build up a solution for $A[n]$ from smaller subproblems?
Suppose we know Split(k) for $k = 0, 1, \ldots, n-1$ where:

$$Split(k) = \begin{cases} True \text{ if } A[1..k] \text{ is splittable} \\ False \text{ otherwise} \end{cases}$$

How do we find Split(n)?
Try Split(j) **and** Word($j+1, n$) for all $j = 0, \ldots n-1$.

Correctness: Split(n) (is true) $\iff$ at least one j gives True.
$\Leftarrow$ Some j exists such that Split(j) **and** Word($j+1, n$) is True.
So, we have a way to split $A[1..n]$.

$\Rightarrow$ If $A[1..n]$ is splittable, take $A[j+1..n]$ as last word.

Text Segmentation

We can then create an algorithm to solve the smaller subproblems and store the results:

```
1. $Split[0] \leftarrow$ True
2. for $k \leftarrow 1$ to $n$ do
3.     $Split[k] \leftarrow$ False
4.     for $j \leftarrow 0$ to $k-1$ do
5.         if $Split[j]$ and $Word(j+1, k)$ then
6.             $Split[k] \leftarrow$ True
```

Runtime: $O(n^2)$
Exercise: Show how to compute the actual split.

Longest Increasing Subsequence

Problem

Longest Increasing Subsequence
Instance: *A sequence of numbers* $A[1..n]$ *where* $A[i] \in \mathbb{N}$*.*
Find: *The longest increasing subsequence (length and/or sequence).*

Example: A : 5 2 1 4 3 1 6 9 2
An increasing subsequence of length 4 is: 5 2 1 4 3 1 6 9 2

Try similar approach to the previous problem.

Let $LIS[k] =$ length of longest increasing subsequence of $A[1..k]$.

Longest Increasing Subsequence

Problem

Longest Increasing Subsequence
Instance: *A sequence of numbers $A[1..n]$ where $A[i] \in \mathbb{N}$.*
Find: *The longest increasing subsequence (length and/or sequence).*

Example: A : 5 2 1 4 3 1 6 9 2
An increasing subsequence of length 4 is: 5 2 1 4 3 1 6 9 2

Try similar approach to the previous problem.

Let $LIS[k]$ = length of longest increasing subsequence of $A[1..k]$.

Not enough information to find $LIS[n]$ - length alone is not enough, need to know last number of subsequence to see if it can be extended by adding $A[n]$ or not.

Longest Increasing Subsequence

Define $LISe[k]$ = length of the longest increasing subsequence of $A[1..k]$ that ends with $A[k]$.

To compute $LISe[k]$, consider all the previous longest sequences that can be extended by $A[k]$.

Longest Increasing Subsequence

Define $LISe[k]$ = length of the longest increasing subsequence of $A[1..k]$ that ends with $A[k]$.

To compute $LISe[k]$, consider all the previous longest sequences that can be extended by $A[k]$.

```
1.  $LISe[1] \leftarrow 1$
2.  **for** $k \leftarrow 2$ **to** $n$ **do**
3.      $LISe[k] \leftarrow 1$
4.      **for** $j \leftarrow 1$ **to** $k-1$ **do**
5.          **if** $A[k] > A[j]$ **then**
6.              $LISe[k] \leftarrow max\{LISe[k], LISe[j]+1\}$
```

Runtime: $O(n^2)$
Exercise: Argue correctness

Longest Increasing Subsequence

Given $LISe[1..n]$, how do you find the maximum length?

Longest Increasing Subsequence

Given $LISe[1..n]$, how do you find the maximum length?

- Find maximum entry in $LISe$, OR
- Add dummy entry $A[n+1] = +\infty$, then return $LISe[n+1] - 1$

How do we recover the actual sequence itself?

- Need to also store which sequence j we extended by adding $A[k]$. Can then backtrack to recover the terms.

Runtime (simple approach): $O(n^2)$ but $O(n \log n)$ is possible.

Longest Common Subsequence

Problem

Longest Common Subsequence
Instance: *Two strings $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$.*
Find: *The longest common subsequence (common to x and y).*

Let $M(i, j)$ = length of longest common subsequence of $x_1 \ldots x_{i-1}x_i$ and $y_1 \ldots y_{j-1}y_j$.

How do we solve a subproblem using “smaller” subproblems?
What are the possibilities?

- Match x_i with y_j, $x_i = y_j$
- Skip x_i
- Skip y_j

Longest Common Subsequence

Base cases: $M(i,0) = 0$ and $M(0,j) = 0$

Recall

- Match x_i with y_j, $x_i = y_j$
- Skip x_i
- Skip y_j

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

Solve subproblems in any order with $M(i-1,j-1)$, $M(i-1,j)$, $M(i,j-1)$ before $M(i,j)$.

Longest Common Subsequence

- Pseudocode for solving the subproblems
- Identifying the Optimal solution
- Recovering the actual subsequence - sometimes you may need to explicitly store the decision made while solving the subproblems.

  For example, store which subproblem was the max value used to compute $M(i, j)$.

Maximum Common Subsequence and Longest Increasing Subsequence

Maximum common subsequence solves longest increasing subsequence.

Claim: Longest increasing subsequence of A = maximum common subsequence of A and S_A where S_A is $sort(A)$.

A : 5 *2* 9 6 *3* *7* 4

S_A : *2* *3* 4 5 6 *7* 9

Designing Strategy for Optimization Problems

Optimal Structure Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain *subproblems* of I.

Define Subproblems Define a set of subproblems $\mathcal{S}(I)$ of the instance I, the solution of which enables the optimal solution of I to be computed. I will be the last or largest instance in the set $\mathcal{S}(I)$.

Designing Strategy for Optimization Problems

Recurrence Relation Derive a recurrence relation on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.

Compute Optimal Solutions Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. Compute these solutions using the recurrence relation in a *bottom-up* fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to I.

Edit Distance

Maximum Common Subsequence found a sequence of characters common to each given string but allowed characters to be skipped over.

Another idea is to count the number of changes it would take to modify one string into the other.

A change is one of:

- add a letter (gap)
- delete a letter (gap)
- replace a letter (mismatch found)

This is called **Edit Distance**.
Used in

Edit Distance

Maximum Common Subsequence found a sequence of characters common to each given string but allowed characters to be skipped over.

Another idea is to count the number of changes it would take to modify one string into the other.

A change is one of:

- add a letter (gap)
- delete a letter (gap)
- replace a letter (mismatch found)

This is called **Edit Distance**.
Used in natural language processing, bioinformatics for comparing DNA sequences, etc; i.e. strings over $\Sigma = \{A, C, T, G\}$.

Edit Distance

Problem

Edit Distance

Instance: *Two strings $x = x_1 \dots x_m$ and $y = y_1 \dots y_n$.*

Find: *The edit distance between x and y; i.e. find the alignment that gives the minimum number of changes.*

Subproblem: $M(i, j) =$ minimum number of changes to match $x_1 \dots x_{i-1}x_i$ and $y_1 \dots y_{j-1}y_j$.

Possible changes:

- match x_i to y_i at a replacement cost if characters are different ($x_1 \neq y_j$)
- match x_i to gap/blank character (delete x_i)
- match y_j to gap/blank character (add y_j)

Note: each change may have a different cost.

Edit Distance

Recurrence relation:

$$M(i,j) = min\begin{cases} M(i-1,j-1) & \text{if if } x_i = y_j \\ r + M(i-1,j-1) & \text{if } x_i \neq y_j \\ d + M(i-1,j) & \text{match } x_i \text{ to blank} \\ a + M(i,j-1) & \text{match } y_j \text{ to blank} \end{cases}$$

where r is the replacement cost, d delete cost and a add cost.
Count the number of changes: $r = d = a = 1$

May be much more sophisticated: replacement cost, $r(x_i, y_j)$, may depend on the letters. For example,
$r(a, s) = 1$ because keys are close on keyboard
$r(a, c) = 2$ because a bit farther away
$r(a, e) = 1$ because both are vowels, etc.

Edit Distance

- Order to solve subproblems
- Pseudocode to solve subproblems
- Optimal solutions
- Recovering the actual changes made
- Runtime and space

Weighted Interval Scheduling

Problem

Interval Scheduling
Instance: *A set of intervals I.*
Find: *A maximum size subset of disjoint intervals.*

Problem

Weighted Interval Scheduling
Instance: *A set of intervals I and weights $w(i)$ for each $i \in I$.*
Find: *A set $S \subseteq I$ such that no two intervals overlap and $\sum_{i \in S} w(i)$ is maximized.*

Intervals: start time, finish time, length, etc
"Weight" could represent many things: profit, preference, etc

Maximum Weight Independent Set

A more general version of this problem can be defined as:

Problem

Maximum Weight Independent Set

Instance: *A set of elements I, weights $w(i)$ for each $i \in I$ and a set C of conflicts where $(i, j) \in C$ if elements i and j conflict.*

Find: *A maximum weight subset $S \subseteq I$ with no conflicting pairs of items.*

Can be modeled as a graph where each element is a vertex and an edge represents a conflict between adjacent elements.

Find: a maximum weight independent set (NP-complete).

General approach: consider element i, either choose it or not.
$OPT(I) = max\{OPT(I - \{i\}), w(i) + OPT(I')\}$ where $I' = \{j | (i, j) \notin C\}$
$T(n) = 2T(n-1) + O(1) \Rightarrow T(n) \in O(2^n)$
May end up solving subproblems for each of the 2^n subsets of I.

Back to Weighted Interval Scheduling

Subproblems: Let $M(i)$ = max weight subset of intervals 1..i.

We can either choose interval i or not.

$$M(i) = max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(X) & \text{if we choose } i \end{cases}$$

Back to Weighted Interval Scheduling

Subproblems: Let $M(i) =$ max weight subset of intervals 1..i.

We can either choose interval i or not.

$$M(i) = max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(X) & \text{if we choose } i \end{cases}$$

Want X to be the intervals that disjoint from i but also to be labelled less than i (so they are "smaller" subproblems).
Can we somehow order the intervals?

Back to Weighted Interval Scheduling

Subproblems: Let $M(i)$ = max weight subset of intervals 1..i.

We can either choose interval i or not.

$$M(i) = max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(X) & \text{if we choose } i \end{cases}$$

Want X to be the intervals that disjoint from i but also to be labelled less than i (so they are "smaller" subproblems).
Can we somehow order the intervals? Yes, call this set $p(i)$.

$$M(i) = max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(p(i)) & \text{if we choose } i \end{cases}$$

Weighted Interval Scheduling

The algorithm to compute the actual set and weight is:

```
1.  Sort intervals 1..$n$ by right endpoint and relabel
2.  $M(0) \leftarrow 0$
3.  $S(0) \leftarrow \emptyset$ // stores set of chosen intervals
4.  for $i \leftarrow 1$ to $n$ do
5.      $p(i) \leftarrow i - 1$ // compute $p(i)$
6.      while $p(i) \neq 0$ and intervals $i$ and $p(i)$ overlap do
7.          $p(i) \leftarrow p(i) - 1$

8.      if $M(i-1) \geq w(i) + M(p(i))$ then
9.          $M(i) \leftarrow M(i-1)$
10.         $S(i) \leftarrow S(i-1)$
11.     else
12.         $M(i) \leftarrow w(i) + M(p(i))$
13.         $S(i) \leftarrow \{i\} \cup S(p(i))$
```

Optimal solution: weight $M(n)$, interval set $S(n)$

Weighted Interval Scheduling

Runtime:
$O(n \log n)$ to sort n subproblems, each $O(n) \Rightarrow O(n^2)$

Space: $O(n^2)$ to store n sets of size $O(n)$

Improvements

- Compute all $p(i)$ values first to save time.
- Compute S by backtracking to save space.

Constructing Optimal Binary Search Trees

Problem

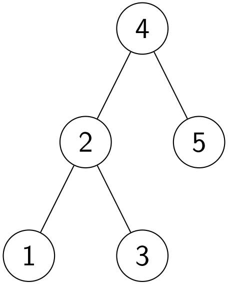
Constructing an Optimal Binary Search Tree
Instance: *A set of items* $I = \{1, \ldots, n\}$ *and probabilities* $p_1, \ldots, p_n$ *where* p_i *is the probability that item* i *will be searched.*
Find: *A BST that minimizes the search cost* $\displaystyle\sum_{i \in I} (p_i) \cdot ProbeDepth(i)$.

$ProbeDepth(i) = 1 + Depth(i)$

The root node has $Depth = 0$ but $ProbeDepth = 1$; i.e. it takes 1 probe to reach it (similar to hashing).

Constructing Optimal Binary Search Trees

For example: $p_1 = p_2 = p_3 = p_4 = p_5 = \frac{1}{5}$

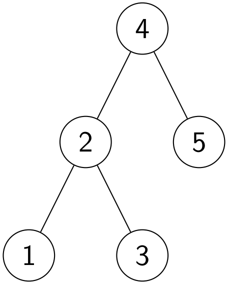


Search Cost $= 1 \cdot 1 \cdot \frac{1}{5} + 2 \cdot 2 \cdot \frac{1}{5} + 2 \cdot 3 \cdot \frac{1}{5} = \frac{7}{5}$

Is the tree unique?

Constructing Optimal Binary Search Trees

For example: $p_1 = 0.6, p_2 = p_3 = p_4 = p_5 = 0.1$

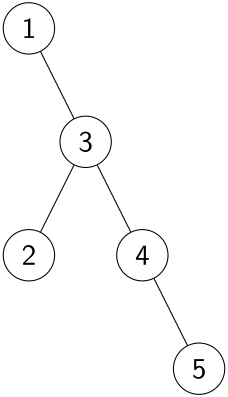


Search Cost $= 1 \cdot 1 \cdot (0.1) + 2 \cdot 2 \cdot (0.1) + 1 \cdot 3 \cdot (0.6) + 1 \cdot 3 \cdot (0.1) = 2.6$

Does this tree minimize the search cost?

Constructing Optimal Binary Search Trees

For example: $p_1 = 0.6, p_2 = p_3 = p_4 = p_5 = 0.1$
Should place the item with the highest probability at the top.



Search Cost $= 1 \cdot 1 \cdot (0.6) + 1 \cdot 2 \cdot (0.1) + 2 \cdot 3 \cdot (0.1) + 1 \cdot 4 \cdot (0.1) = 1.8$

Does this tree minimize the search cost?

Constructing Optimal Binary Search Trees

Dynamic Programming approach: Try all choices for root node, ...

- Suppose root will be some item k.
- Left subtree is then the optimal BST on $1, \ldots, k-1$.
- Right subtree is then the optimal BST on $k+1, \ldots, n$.

Constructing Optimal Binary Search Trees

Dynamic Programming approach: Try all choices for root node, ...

- Suppose root will be some item k.
- Left subtree is then the optimal BST on $1, \dots, k-1$.
- Right subtree is then the optimal BST on $k+1, \dots, n$.

Subproblems: Let $M[i,j]$ be the optimal BST on items $i, \dots, j$.

$$M[i,j] = \min_{k=i..j} \Big\{ M[i, k-1] + M[K+1, j] \Big\} + \sum_{t=i}^{j} p_t$$

- One of the nodes $k \in i, \dots, j$ will be the root so contributes $1 \cdot 1 \cdot p_k$.
- $M[i, k-1]$ gives the search cost for this tree but doesn't consider it is the left subtree of k so we need to add $\sum_{t=i}^{k-1} p_t$.
- Similarly, for the right subtree, we need to add $\sum_{t=k+1}^{j} p_t$.

Constructing Optimal Binary Search Trees

Let $P[i] = \sum_{t=1}^{i} p_t$ where $P[0] = 0$, so, $\sum_{t=i}^{j} p_t = P[j] - P[i-1]$.

```
1.  for $i \leftarrow 1$ to $n$ do
2.      $M[i,i] \leftarrow p_i$ // Single node tree
3.      $M[i,i-1] \leftarrow 0$ // empty tree
4.  for $d \leftarrow 1$ to $n-1$ do // $d = j - i$ from above
5.      for $i \leftarrow 1$ to $n-1$ do // Find $M[i, i+d]$
6.          $best \leftarrow \infty$
7.          for $k \leftarrow i$ to $i+d$ do
8.              $temp \leftarrow M[i,k-1] + M[k+1,i+d]$
9.              if $temp < best$ then $best \leftarrow temp$
10.         $M[i,i+d] \leftarrow best + P[i+d] - P[i-1]$
```

Runtime: $O(n^2 \cdot n) = O(n^3)$

0-1 Knapsack

Problem

0-1 Knapsack

Instance: *A set of items $\{1, \ldots, n\}$ where item i has weight w_i and value v_i and a knapsack with capacity W.*

Find: *A subset of items S such $\sum_{i \in S} w_i \leq W$ so that $\sum_{i \in S} v_i$ is maximized.*

Note: "0-1"- you must take the whole item or none of it; items are not divisible (Fractional Knapsack is a different problem).

Dynamic Programming Approach:

0-1 Knapsack

Problem

0-1 Knapsack

Instance: *A set of items $\{1, \ldots, n\}$ where item i has weight w_i and value v_i and a knapsack with capacity W.*

Find: *A subset of items S such $\sum_{i \in S} w_i \leq W$ so that $\sum_{i \in S} v_i$ is maximized.*

Note: "0-1"- you must take the whole item or none of it; items are not divisible (Fractional Knapsack is a different problem).

Dynamic Programming Approach:

Consider items $1, \ldots, i$, *is item i in or out?*.

- If $i \notin S \Rightarrow$ Optimal solution on $1, \ldots, i-1$
- If $i \in S \Rightarrow$ If we take i, what subproblem do we want?

0-1 Knapsack

Problem

0-1 Knapsack

Instance: *A set of items $\{1, \ldots, n\}$ where item i has weight w_i and value v_i and a knapsack with capacity W.*

Find: *A subset of items S such $\sum_{i \in S} w_i \leq W$ so that $\sum_{i \in S} v_i$ is maximized.*

Note: "0-1"- you must take the whole item or none of it; items are not divisible (Fractional Knapsack is a different problem).

Dynamic Programming Approach:

Consider items $1, \ldots, i$, *is item i in or out?*.

- If $i \notin S \Rightarrow$ Optimal solution on $1, \ldots, i-1$
- If $i \in S \Rightarrow$ If we take i, what subproblem do we want?
  - Maximize $\sum$ *values* considering items $1, \ldots, i-1$
  - Reduced weight (capacity left after taking i): $\sum$ *weight* $\leq W - w_i$

0-1 Knapsack

- Define subproblems (i, w) where $i = 0..n$ and $w = 0..W$
- Give pseudocode to solve subproblems in appropriate order.
- Identify optimal solution - maximized value.
- Recover actual items chosen.
- Analysis: Pseudo-polynomial runtime.

Memoization

Memoization (optimization technique): store the result of expensive function calls and return the stored result instead of recomputing.

- Use recursion, rather than explicitly solving all subproblems bottom-up (as we have done so far).
- *Danger!* Don't want to solve the same subproblem over and over (possibly taking exponential time; e.g. $T(n) = 2T(n-1) + O(1)$ is exponential).

Memoization

Memoization (optimization technique): store the result of expensive function calls and return the stored result instead of recomputing.

- Use recursion, rather than explicitly solving all subproblems bottom-up (as we have done so far).
- *Danger!* Don't want to solve the same subproblem over and over (possibly taking exponential time; e.g. $T(n) = 2T(n-1) + O(1)$ is exponential).
- Fix: when you solve a subproblem, store the solutions. Before (re)solving a problem, check if you have already stored the solution. Solutions can be stored in a matrix or in a hash table.
  Some programming languages will help you implement memoization:
  - `memoized-call(factorial(n))` in Python
  - `option remember` in Maple, etc.

Memoization

Advantage

- Maybe don't have to solve **all** the subproblems.

Disadvantages

- Harder to analyze runtime.
- Recursion adds extra overhead - runtime stack, etc.

Single Source Shortest Paths in a DAG

A *directed acyclic graph* (DAG) has no directed cycle.

Idea: Use topological sort $v_1 v_2 \dots v_n$ so every edge (v_i, v_j) has $i < j$.

If v comes before s, there is no path $s \to v$ so remove all such vertices, relabel, let $s = v_1$.

```
1.  $d_i \leftarrow \infty \ \forall i$
2.  $d_1 \leftarrow 0$
3.  for $i$ from 1 to $n$ do
4.      for every edge $(v_i, v_j)$ do
5.          if $d_i + w(v_i, v_j) < d_j$ then
6.              $d_j \leftarrow d_i + w(v_i, v_j)$
```

Analysis: $O(n + m)$

Claim: This finds shortest paths from s.
Exercise: Proof by induction on i.

Dynamic Programming for Shortest Paths in a Graph

Single Source Shortest Paths: Bellman-Ford

- The original application of dynamic programming.
- Edge weights may be negative but no negative weight cycles.

All Pairs Shortest Paths: Floyd-Warshall

How do we define a subproblem?

Consider a uv path that goes through a vertex x.

Dynamic Programming for Shortest Paths in a Graph

Single Source Shortest Paths: Bellman-Ford

- The original application of dynamic programming.
- Edge weights may be negative but no negative weight cycles.

All Pairs Shortest Paths: Floyd-Warshall

How do we define a subproblem?

Consider a uv path that goes through a vertex x.
$\Rightarrow$ Consists of the shortest ux path $+$ shortest xv path.

In what sense do we consider these “smaller”?

- Fewer edges: try paths of ≤ 1 edge, $\leq 2 edges$, etc.
  We will use this for the single source shortest paths algorithm.
- They don’t use x.
  We will use this for all pairs shortest paths algorithm.

Single Source Shortest Paths

Let $d_i(v)$ be the weight of the shortest path from s to v using $\leq i$ edges. Then,

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(s, v) & \text{if } (s, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

and we want to find $d_{n-1}(v)$.

Why $n - 1$ edges?

Single Source Shortest Paths

Let $d_i(v)$ be the weight of the shortest path from s to v using $\leq i$ edges. Then,

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

and we want to find $d_{n-1}(v)$.

Why $n-1$ edges?

- A path with $\geq n$ edges would repeat a vertex giving a cycle.
- Every cycle has weight ≥ 0, removing the cycle is no worse.

Single Source Shortest Paths

Let $d_i(v)$ be the weight of the shortest path from s to v using $\leq i$ edges. Then,

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

and we want to find $d_{n-1}(v)$.

Why $n-1$ edges?

- A path with $\geq n$ edges would repeat a vertex giving a cycle.
- Every cycle has weight ≥ 0, removing the cycle is no worse.

Compute d_i from d_{i-1}:

$$d_i(v) = min \begin{cases} d_{i-1}(v) & \text{use } \leq i-1 \text{ edges} \\ min_u\{d_{i-1}(u) + w(u,v)\} & \text{use } i \text{ edges} \\ \infty & \text{otherwise} \end{cases}$$

Correctness: We consider all possibilities for d_i. Use induction on i.

Bellman-Ford Algorithm

```
1.  Initialize $d_1(v)$ for all $v$ (previous slide)
2.  for $i$ from 2 to $n-1$ do
3.      for $v \in V$ do
4.          $d_i(v) \leftarrow d_{i-1}(v)$
5.          for each edge $(u, v)$ do // Want edges directed into $v$
6.              $d_i(v) \leftarrow min\{d_i(v), d_{i-1}(u) + w(u, v)\}$
```

Analysis: $O(n \cdot (n+m))$

Save space, re-use same $d(v)$, and simplify code:

```
1.  $d(v) \leftarrow \infty$ for all $v$
2.  $d(s) \leftarrow 0$
3.  for $i$ from 1 to $n-1$ do
4.      for $(u, v) \in E$ do
5.          $d(v) \leftarrow min\{d(v), d(u) + w(u, v)\}$
```

Bellman-Ford Algorithm

Exercise: Note that in the simplified code, i does not appear inside the loop. Explain why the simplified code works; i.e. does the same as the original.

Exercise: If the inner for loop completes and no $d(v)$ value has changed during its execution, justify that the outer loop may exit early.

Exercise: Enhance the code to find the actual shortest paths by adding parent pointers and updates (when d is updated).
Note: paths are recovered would then be in reverse order.

Exercise: If we run 1 more iteration and see if any d value changes, we can detect negative weight cycles reachable from s.
Explain why this works.

Exercise: Show how to detect a negative weight cycle anywhere in the graph.
Solution: add new s' and add edges $(s', v) \forall v$, with weight 0.

All Pairs Shortest Paths

Problem

All Pairs Shortest Paths
Instance: *A directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$ (but no negative weight cycle).*
Find: *The shortest path from u to v for all u, v.*
Output the distances as an $n \times n$ matrix $D[u, v]$.

Idea: Use dynamic programming where intermediate paths use only a subset of the vertices.

Let $V = \{1, 2, \ldots, n\}$.
Let $D_i[u, v]$ be the length of the shortest uv path using intermediate vertices in $\{1, 2, \ldots, i\}$.

Subproblems: Solve $D_i[u, v]$ for all u, v as i goes from 0 to n.
Our final solution is then $D_n[u, v]$.

Recurrence Relation for All Pairs Shortest Paths

Recall: $D_i[u, v]$ is the length of the shortest uv path using intermediate vertices in $\{1, 2, \ldots, i\}$.

Base cases:

$$D_0[u, v] = \begin{cases} 0 & \text{if } u = v \\ w(u, v) & \text{use } (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

The main recursive property, $i > 0$, is then to use i or not:

$$D_i[u, v] = min\begin{cases} D_{i-1}[u, i] + D_{i-1}[i, v] & \text{use vertex } i \\ D_{i-1}[u, v] & \text{don't use } i \end{cases}$$

Correctness: We consider all possibilities for D_i. Use induction on i.

Floyd-Warshall Algorithm

```
1.  Initialize $D_0[u, v]$ (Base cases on previous slide)
2.  **for** $i$ **from** 1 **to** $n$ **do**
3.      **for** $u$ **from** 1 **to** $n$ **do**
4.          **for** $v$ **from** 1 **to** $n$ **do**
5.              $D_i[u, v] \leftarrow min\{D_i[u, v], D_{i-1}[u, i] + D_{i-1}[i, v]\}$
```

Analysis: $O(n^3)$ time and $O(n^3)$ space

Exercise: Give the intialization for D and explain why the following algorithm that reduces the space requirement to $O(n^2)$ is also correct:

```
1.  **for** $i$ **from** 1 **to** $n$ **do**
2.      **for** $u$ **from** 1 **to** $n$ **do**
3.          **for** $v$ **from** 1 **to** $n$ **do**
4.              $D[u, v] \leftarrow min\{D[u, v], D[u, i] + D[i, v]\}$
```

Recovering the Actual Path

Create a new array $Next[u, v]$ where each location stores the first vertex after u on a shortest path from u to v.

Suppose: a shortest uv path follows: u, x, y, z, v.
Then, $Next[u, v]$ returns x, $Next[x, v]$ returns y, $Next[y, v]$ returns z.

Implementation:
When we update $D[u, v] \leftarrow min\{D[u, v], D[u, i] + D[i, v]\}$,
also update $Next[u, v]$ (Exercise).

History: Bellman explains the reasoning behind the term dynamic programming in his autobiography, Eye of the Hurricane: An Autobiography:
`https://en.wikipedia.org/wiki/Dynamic_programming#History`