CS3000: Algorithms & Data
Paul Hand

Lecture 22:

- Review

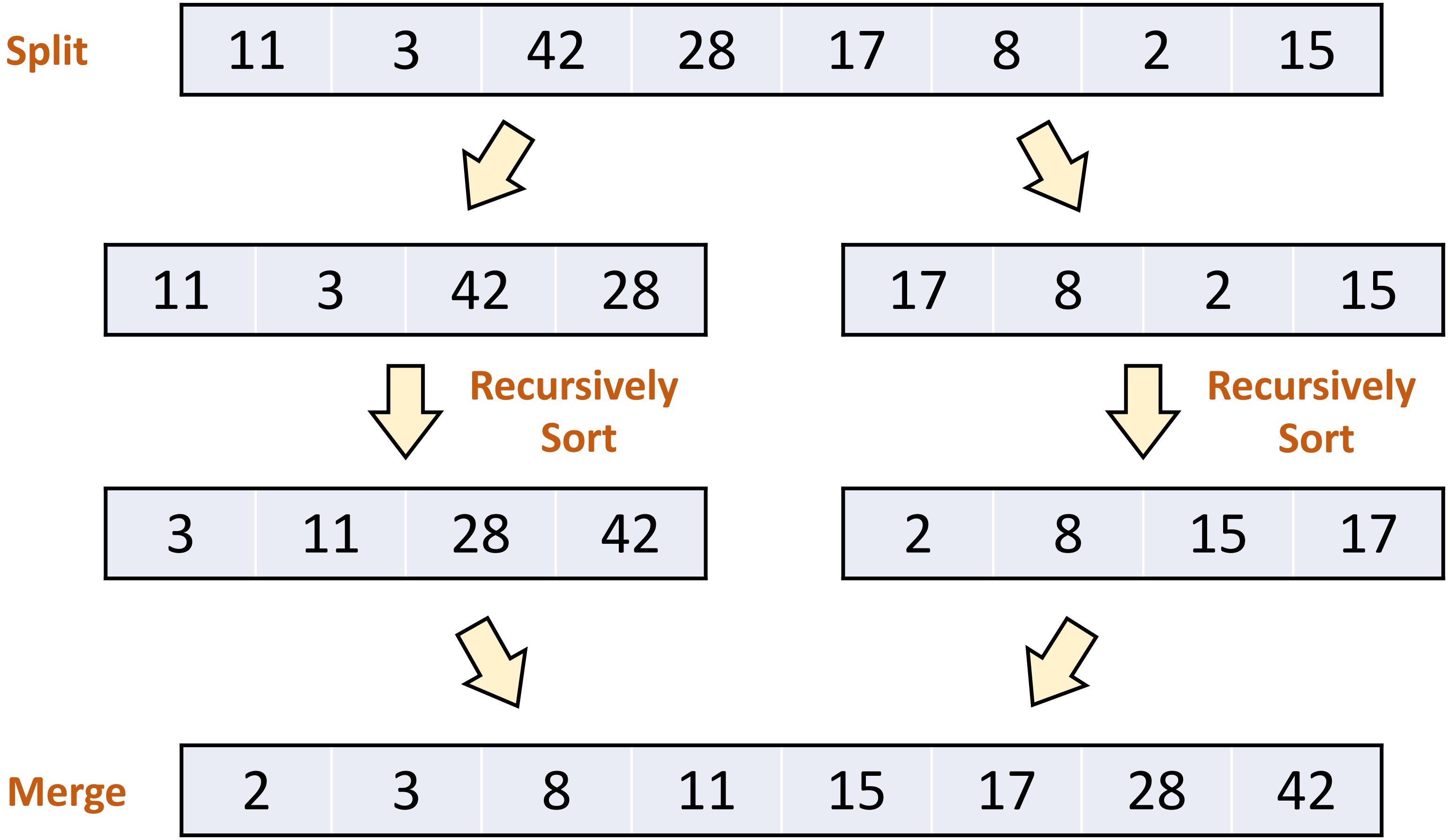
Apr 18, 2019

	$\lim_{n \to \infty} f/g$
$f = O(g)$	$< \infty$
$f = \Omega(g)$	> 0
$f = \Theta(g)$	$< \infty$ & > 0
$f = o(g)$	$= 0$
$f = \omega(g)$	$= \infty$

Divide and Conquer Algorithms

- **Examples:**
  - Mergesort: sorting a list
  - Binary Search: search in a sorted list
  - Karatsuba’s Algorithm: integer multiplication
  - Closest pair of points
  - Fast Fourier Transform
  - …
- **Key Tools:**
  - Correctness: proof by induction
  - Running Time Analysis: recurrences
  - Asymptotic Analysis

Divide and Conquer: Mergesort



Merging two sorted lists

```
Merge(L,R):
  Let n ← len(L) + len(R)
  Let A be an array of length n
  j ← 1, k ← 1,

  For i = 1,…,n:
    If (j > len(L)):          // L is empty
      A[i] ← R[k], k ← k+1
    ElseIf (k > len(R)):      // R is empty
      A[i] ← L[j], j ← j+1
    ElseIf (L[j] <= R[k]):    // L is smallest
      A[i] ← L[j], j ← j+1
    Else:                     // R is smallest
      A[i] ← R[k], k ← k+1

  Return A
```

- **Prove:** If L and R are sorted from smallest to largest, then A is sorted from smallest to largest.

MergeSort Algorithm

```
MergeSort(A):
  If (len(A) = 1): Return A    // Base Case

  Let m ← ⌈len(A)/2⌉           // Split
  Let L ← A[1:m], R ← A[m+1:n]

  Let L ← MergeSort(L)         // Recurse
  Let R ← MergeSort(R)

  Let A ← Merge(L,R)           // Merge

  Return A
```

Mergesort Summary

- Sort a list of n numbers in $\Theta(n \log_2 n)$ time
  - Can actually sort anything that allows comparisons
  - No comparison based algorithm can be (much) faster
- Divide-and-conquer
  - Break the list into two halves, sort each one and merge
  - Key Fact: Merging sorted lists is easier than sorting
- Proof of correctness
  - Proof by induction
- Analysis of running time
  - Recurrences

Recursion Tree

- $T(n) = \boldsymbol{a}T(n/\boldsymbol{b}) + n^{\boldsymbol{d}}$

Level	Size	Work
0	n	n^d
1	n/b ... n/b a calls	$a \times \left(\frac{n}{b}\right)^d = \left(\frac{a}{b^d}\right) \cdot n^d$
2	n/b^2 ... n/b^2 a^2 calls	$a^2 \times \left(\frac{n}{b^2}\right)^d = \left(\frac{a}{b^d}\right)^2 \cdot n^d$
i	n/b^i a^i calls	$\left(\frac{a}{b^d}\right)^i \cdot n^d$
$\log_b n$	1 1 1 $a^{\log_b n}$	$a^{\log_b n} = \left(\frac{a}{b^d}\right)^{\log_b n} \cdot n^d$

Activity:

Where (which level) is the most work happening?

If $\frac{a}{b^d} < 1$, level 0!

If $a/b^d > 1$, level $\log_b n$

If $a/b^d = 1$, all comparable

The "Master Theorem"

- Recipe for recurrences of the form:
  - $T(n) = \boldsymbol{a} \cdot T(n/\boldsymbol{b}) + Cn^{\boldsymbol{d}}$
- Three cases:
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^d}\right) > 1: \; T(n) = \Theta\left(n^{\log_{\boldsymbol{b}} \boldsymbol{a}}\right)$
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^d}\right) = 1: \; T(n) = \Theta\left(n^{\boldsymbol{d}} \log n\right)$
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^d}\right) < 1: \; T(n) = \Theta\left(n^{\boldsymbol{d}}\right)$

Maximum Sum Subarray Problem

- Input: Array A[1:n] of integers
- Problem: Find a subarray A[i:j] with the largest possible sum
- Example: A = [3, -4, 5, -2, -2, 6, -3, 5, -3, 2]

- Task: Devise a divide and conquer algorithm to solve this problem. Consider an algorithm that divides A into two halves.

Discuss w/ neighbors: What is a reasonable place to start attacking this problem.
Find 3 things

- Finding an approach you need to beat
- Study a smaller instance and try to solve it in your head
- consider special cases (Eg pos/neg entries)
- Do you understand problem/vocabulary

Binary Search

anything left is < 28. Dont look at it

Is 28 in this list?

~~2~~	~~3~~	~~8~~	11	15	17	28	42	A

Naive alg: linear search. check $A[i]$ for $i = 1 \ldots n$. $O(n)$ time. Bad

did not exploit structure

~~15~~	17	28	42

28	42

get 28. in list.

Binary Search

```
Search(A,t):
  // A[1:n] sorted in ascending order
  Return BS(A,1,n,t)
```

left end of "active" region

```
BS(A,ℓ,r,t):
```

right end of "active" region

```
  If(ℓ > r): return FALSE
```

width

$$m \leftarrow \ell + \left\lfloor \frac{r-\ell}{2} \right\rfloor$$

midpoint of list & round down

```
  If(A[m] = t): Return m
  ElseIf(A[m] > t): Return BS(A,ℓ,m-1,t)
  Else: Return BS(A,m+1,r,t)
```

nothing to right of m matters

modify right endpoint

Binary Search Wrapup

- Search a sorted array in time $O(\log n)$!!!
- Divide-and-conquer approach
  - Find the middle of the list, recursively search half the list
  - **Key Fact:** eliminate half the list each time
- Prove correctness via induction
- Analyze running time via recurrence
  - $T(n) = T(n/2) + C$

If we want to search many times, worth it to sort in advance

Q: If I want to check if $t \in$ list A, is it worth it to sort A and do binary search??

Sort: $n \log n$ Search: $\log n$ $n \log n + \log n = \Theta(n \log n)$

Dynamic Programming

Dynamic programming is careful recursion

- Break the problem up into small pieces
- Recursively solve the smaller pieces
- Store outcomes of smaller pieces that get called multiple times
- **Key Challenge:** identifying the pieces

Interval Scheduling

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …

- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S maximizing the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \dots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$

A Recursive Formulation

- Let $OPT(i)$ be the **value of the optimal schedule** using only the intervals $\{1, \ldots, i\}$
- **Case 1:** Final interval is not in O $(i \notin O)$
  - Then O must be the optimal solution for $\{1, \ldots, i-1\}$
- **Case 2:** Final interval is in O $(i \in O)$
  - Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  - Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O must be i + the optimal solution for $\{1, \ldots, p(i)\}$

- $OPT(i) = \max\{OPT(i-1), v_n + OPT\big(p(i)\big)\}$
- $OPT(0) = 0, OPT(1) = v_1$

Dynamic Programming Recap

- Express the optimal solution as a **recurrence**
  - Identify a small number of subproblems
  - Relate the optimal solution on subproblems
- Efficiently solve for the **value** of the optimum
  - Simple implementation is exponential time
  - Top-Down: store solution to subproblems
  - Bottom-Up: iterate through subproblems in order
- Find the **solution** using the table of **values**

The Knapsack Problem

- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \dots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T

- **SubsetSum:** $v_i = w_i$

assuming these are natural #s

size

Could write as optimization problem

$$\max_{\substack{S \subseteq \{1 \dots n\} \\ W_S \leq T}} V_S$$

Is there a subset that adds up to T?

Tug of War: $T = \frac{1}{2} \sum_{i=1}^{n} V_i$

Dynamic Programming

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $i \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 and size S
- **Case 2:** $i \in O_{j,S}$
  - Use i + opt. solution for items 1 to j-1 and size $S - w_j$

Recurrence:

$$\mathrm{OPT}(j, S) = \begin{cases} \max\{\overbrace{OPT(j-1, S)}^{\text{case 1}}, \overbrace{v_j + OPT(j-1, S-w_j)}^{\text{case 2}}\} & \text{if } w_j \leq S \\ OPT(j-1, S) & \text{if } w_j > S \end{cases}$$

Can't afford j, not in set

Base Cases:

$$\mathrm{OPT}(j, 0) = \mathrm{OPT}(0, S) = 0$$

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,$] ← 0, M[j,0] ← 0
  for (j = 1,…,n):
    for (s = 1,…,T):
      if (w_j > $): M[j,$] ← M[j-1,$]
      else: M[j] ← max{M[j-1,$],v_j + M[j-1,$-w_j]}

  return M[n,T]
```

base cases

T put 0's in entire row

s s s s ,s s

Activity: What is the runtime of this algorithm?

nT

depends on size of the knapsack

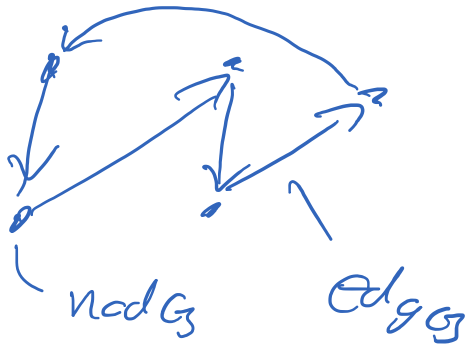
How much memory does it take?

nT

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return ∅
  else:
    if (w_n > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > v_n + M[n-1,T-w_n]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-w_n)
```

Graphs: Key Definitions



Edges are like arrows

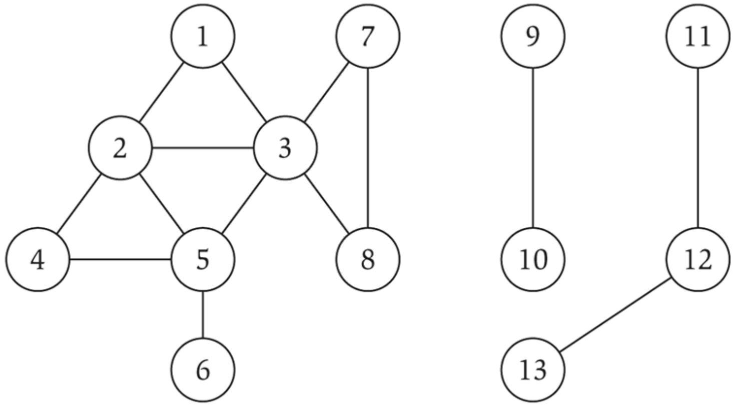
- **Definition:** A directed graph $G = (V, E)$
  - V is the set of nodes/vertices
  - $E \subseteq V \times V$ is the set of edges
  - An edge is an ordered $e = (u, v)$ "from u to v"

set of pairs of vertices

If $(u,v) \in E$, it is an edge of the graph

- **Definition:** An undirected graph $G = (V, E)$
  - Edges are unordered $e = (u, v)$ "between u and v"

- **Simple Graph:**
  - No duplicate edges
  - No self-loops $e = (u, u)$



all one graph

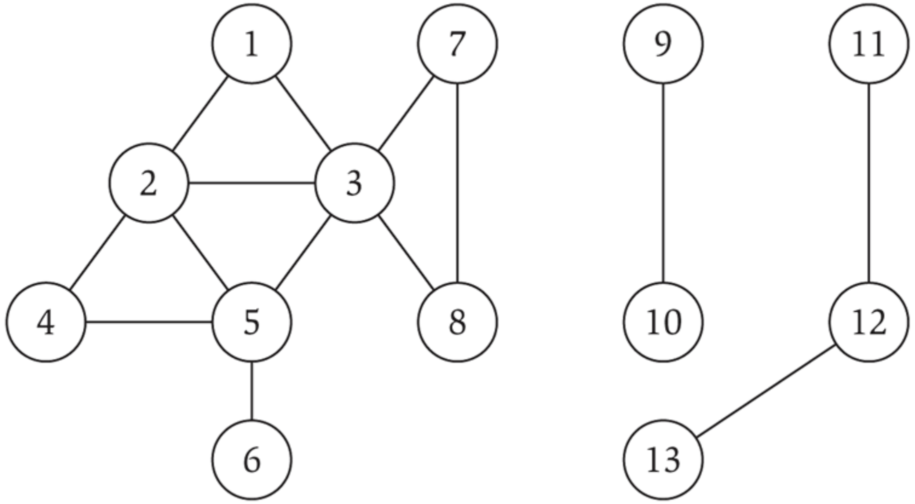
Paths/Connectivity

- A path is a sequence of consecutive edges in E
  - $P = \{(u, w_1), (w_1, w_2), (w_2, w_3), \ldots, (w_{k-1}, v)\}$
  - $P = u - w_1 - w_2 - w_3 - \cdots - w_{k-1} - v$
  - The length of the path is the # of edges

- An undirected graph is connected if for every two vertices $u, v \in V$, there is a path from u to v
- A directed graph is strongly connected if for every two vertices $u, v \in V$, there are paths from u to v and from v to u

Cycles

- A cycle is a path $v_1 - v_2 - \cdots - v_k - v_1$ where $k \geq 3$ and $v_1, \ldots, v_k$ are distinct



Activity: how many cycles are there in this graph?

2-Coloring

- **Problem:** Team Forming
  - Need to form two teams $\boldsymbol{R}, \boldsymbol{P}$
  - Some people don't want to be on the same team as certain other people

Set of people

- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v wont be on the same team
- **Output:** Split V into two sets $\boldsymbol{R}, \boldsymbol{P}$ so that no pair in either set is connected by an edge

root – ... — ... — node – node — ... — ... — root

Designing the Algorithm

- **Claim:** If BFS fails, then G contains an odd cycle
  - If G contains an odd cycle then G can't be 2-colored!

① Within BFS tree, coloring correct

Explore edges not in BFS tree

Case I: non-tree edge is between adjacent layers

Fine for 2 colorability

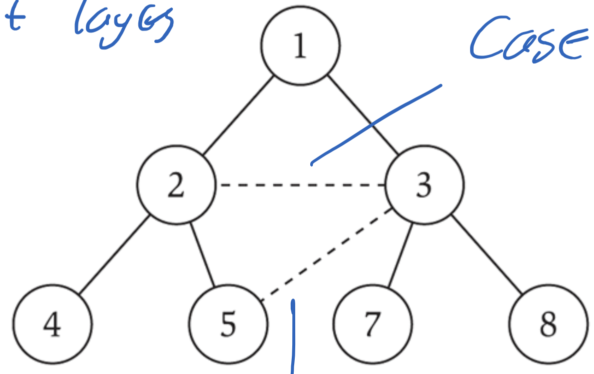
Case II: Edge is within a layer

violates two colorability

We can build odd cycle

root ⋯ node – node ⋯ root

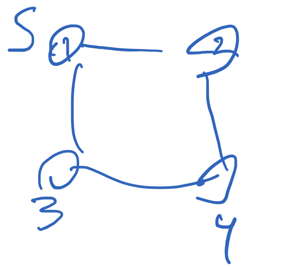
Case II

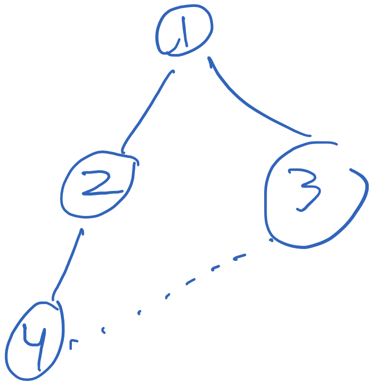


Case I

Above 1 – 2 – 3 – 1

Even cycle

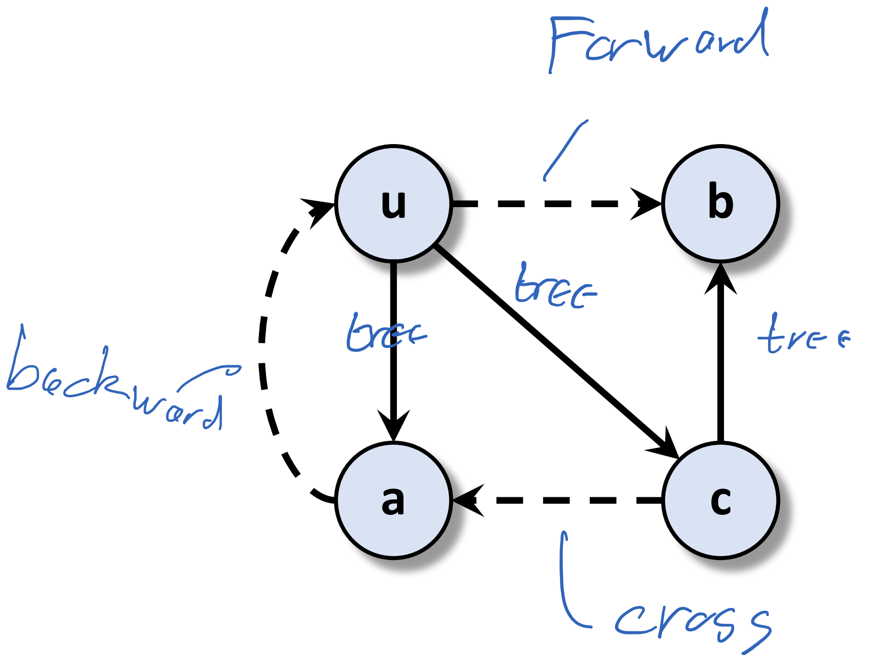




Depth-First Search

DFS tree

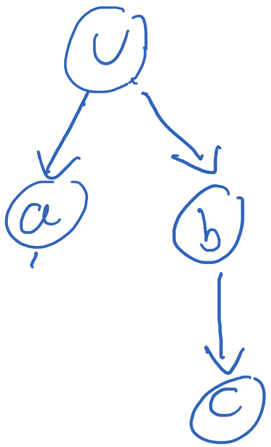
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - **Tree edges:** $(u, a), (u, c), (c, b)$
    - These are the edges that explore new nodes
  - **Forward edges:** (u, b)
    - Ancestor to descendant
  - **Backward edges:** (a, u)
    - Descendant to ancestor
  - **Cross edges:** (c, a)
    - No ancestral relation



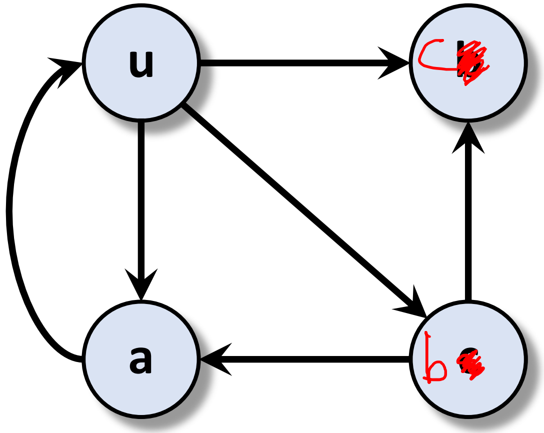
Pre-Ordering

Sorted alphabetically

- Order the vertices by when they were **first** visited by DFS




```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  pre-visit(u)

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
```



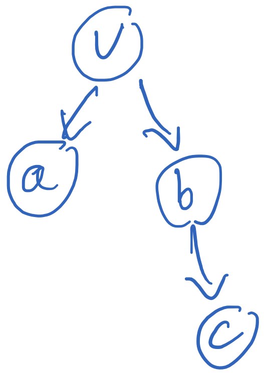
Vertex	Pre-Order
u	1
a	2
b	3
c	4

- Maintain a counter `clock`, initially set `clock = 1`
- `pre-visit(u):`
  `set preorder[u]=clock, clock=clock+1`

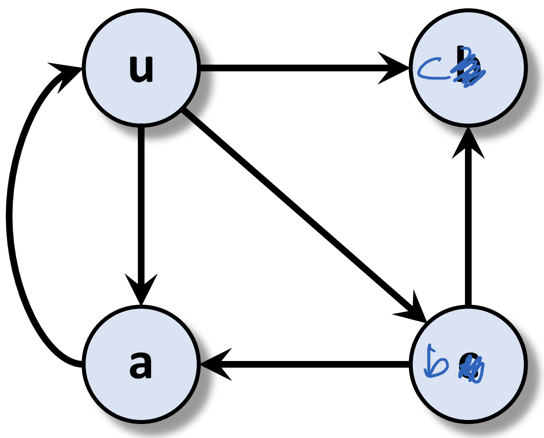
Post-Ordering

- Order the vertices by when they were **last** visited by DFS

We are done processing a node once we process all of its children




```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)

  post-visit(u)
```

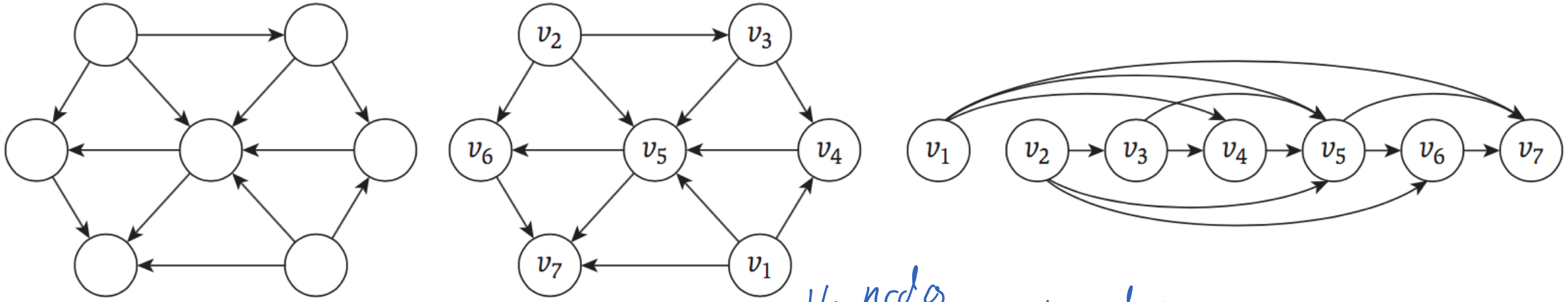


Vertex	Post-Order
u	4
a	1
b	3
c	2

- Maintain a counter `clock`, initially set `clock = 1`
- `post-visit(u):`
  `set postorder[u]=clock, clock=clock+1`

Topological Ordering (TO)

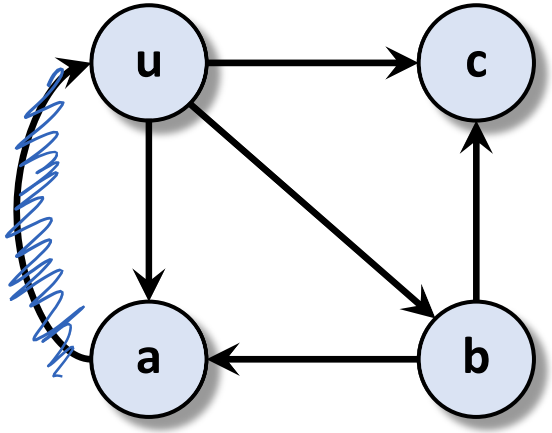
- **DAG:** A directed graph with no directed cycles
- Any DAG can be **toplogically ordered**
  - Label nodes $v_1, \dots, v_n$ so that $(v_i, v_j) \in E \Longrightarrow j > i$



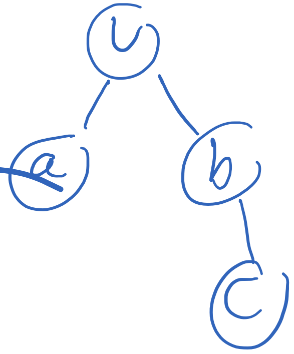
\# node   \# edges

- Can compute a TO in $O(n + m)$ time using DFS
  - Reverse of post-order is a topological order

Algorithm for Topological Ordering

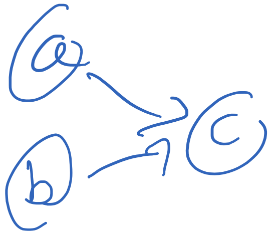


- **Claim:** ordering nodes by decreasing postorder gives a topological ordering
- **Proof:**
  - A DAG has no backward edges (Such an edge would form a cycle)
  - Suppose this is **not** a topological ordering
    - That means there exists an edge (u,v) such that postorder[u] < postorder[v]
    - We showed that any such (u,v) is a backward edge
    - But there are no backward edges, contradiction!



Post order
a, c, b, U

Reversed Postorder
U, b, c a



Shortest Paths

- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

- The distance $d(s, t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s, t \in V$, find the shortest path from s to t
- **Single-Source Shortest Paths:** given a node $s \in V$, find the shortest paths from s to **every** $t \in V$

Structure of Shortest Paths

- If $(u, v) \in E$, then $d(s, v) \leq d(s, u) + \ell(u, v)$ for every node $s \in V$

- If $(u, v) \in E$, and $d(s, v) = d(s, u) + \ell(u, v)$ then there is a shortest $s \leadsto v$-path ending with (u, v)

Weighted Graphs

- **Definition:** A weighted graph $G = (V, E, \{w(e)\})$
  - V is the set of vertices
  - $E \subseteq V \times V$ is the set of edges
  - $w_e \in \mathbb{R}$ are edge weights/lengths/capacities
  - Can be directed or undirected

- **Today:**
  - Directed graphs (one-way streets)
  - Strongly connected (there is always some path)
  - Non-negative edge lengths ($\ell(e) \geq 0$)

Shortest Paths

- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths

- The distance $d(s, t)$ is the length of the shortest path from s to t
- **Shortest Path:** given nodes $s, t \in V$, find the shortest path from s to t
- **Single-Source Shortest Paths:** given a node $s \in V$, find the shortest paths from s to **every** $t \in V$

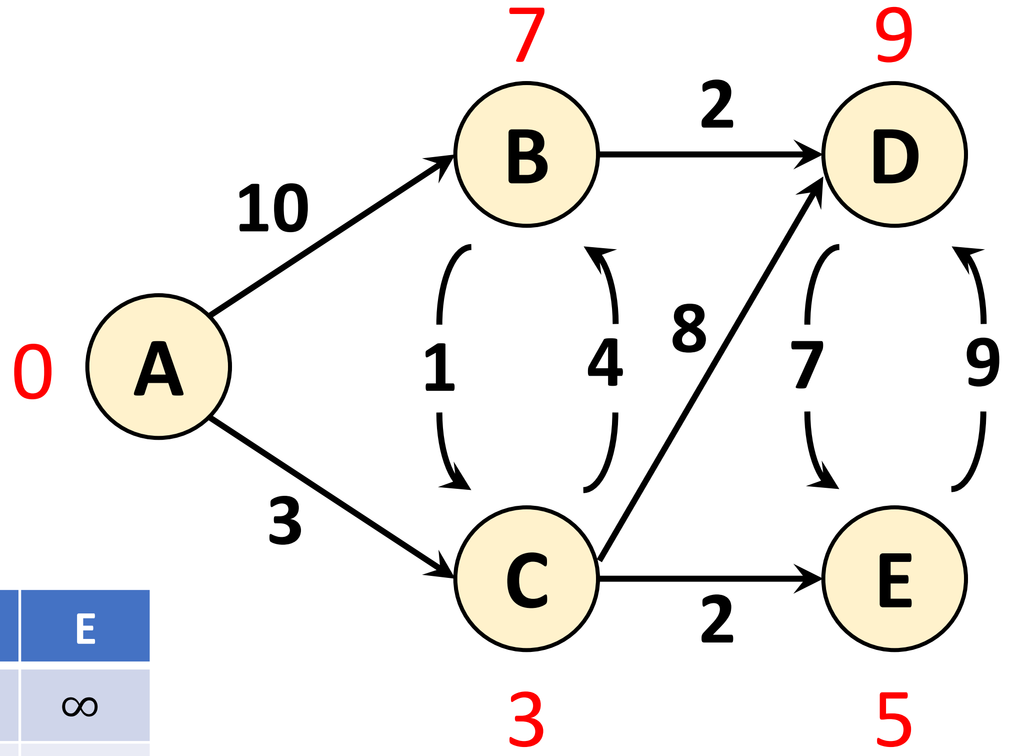
Implementing Dijkstra

```
Dijkstra(G = (V,E,{ℓ(e)}, s):
  d[s] ← 0, d[u] ← ∞ for every u != s
  parent[u] ←⊥ for every u
  Q ← V                    // Q holds the unexplored nodes

  While (Q is not empty):
    u ← argmin_{w∈Q} d[w]  //Find closest unexplored
    Remove u from Q

    // Update the neighbors of u
    For ((u,v) in E):
      If (d[v] > d[u] + ℓ(u,v)):
        d[v] ← d[u] + ℓ(u,v)
        parent[v] ← u

  Return (d, parent)
```

current estimate

Dijkstra's Algorithm: Demo

Don't need to explore D



	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B, D\}$$

Implementing Dijkstra Naively

- Need to explore all n nodes
- Each exploration requires:
  - Finding the unexplored node u with smallest distance
  - Updating the distance for each neighbor of u
    - Lookup current distance
    - Possibly decrease distance

Priority Queues

- Need a data structure Q to hold key-value pairs

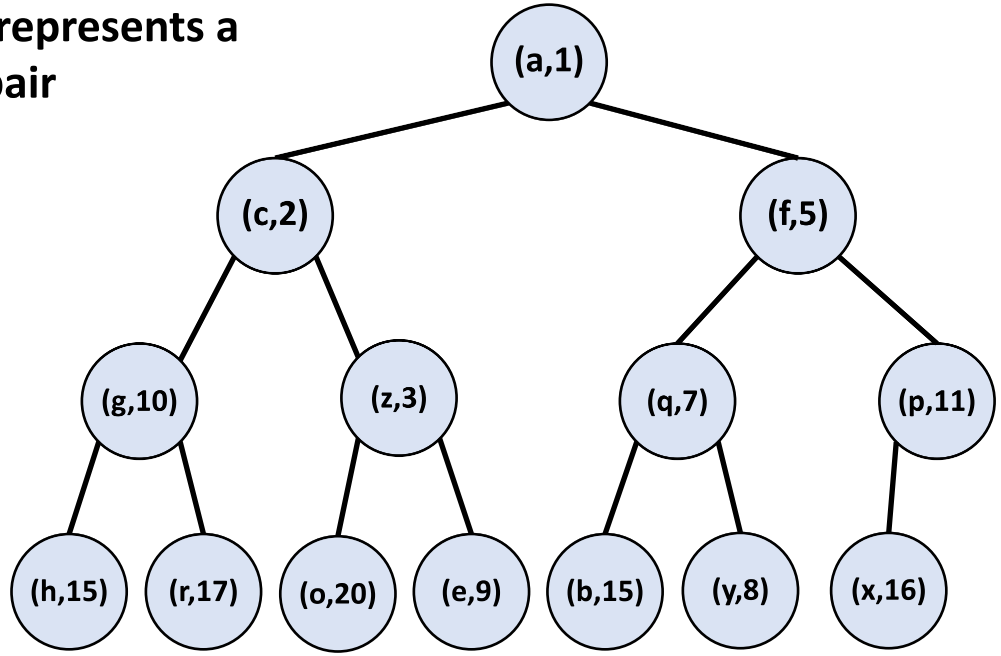
key : node, u
value : d[u]

- Need to support the following operations
  - Insert(Q,k,v): add a new key-value pair
  - Lookup(Q,k): return the value of some key
  - ExtractMin(Q): identify the key with the smallest value
  - DecreaseKey(Q,k,v): reduce the value of some key

Heaps

- **Organize key-value pairs as a binary tree**
  - Later we'll see how to store pairs in an array
- **Heap Order:** If a is the parent of b, then v(a) ≤ v(b)

Each node represents a key-value pair

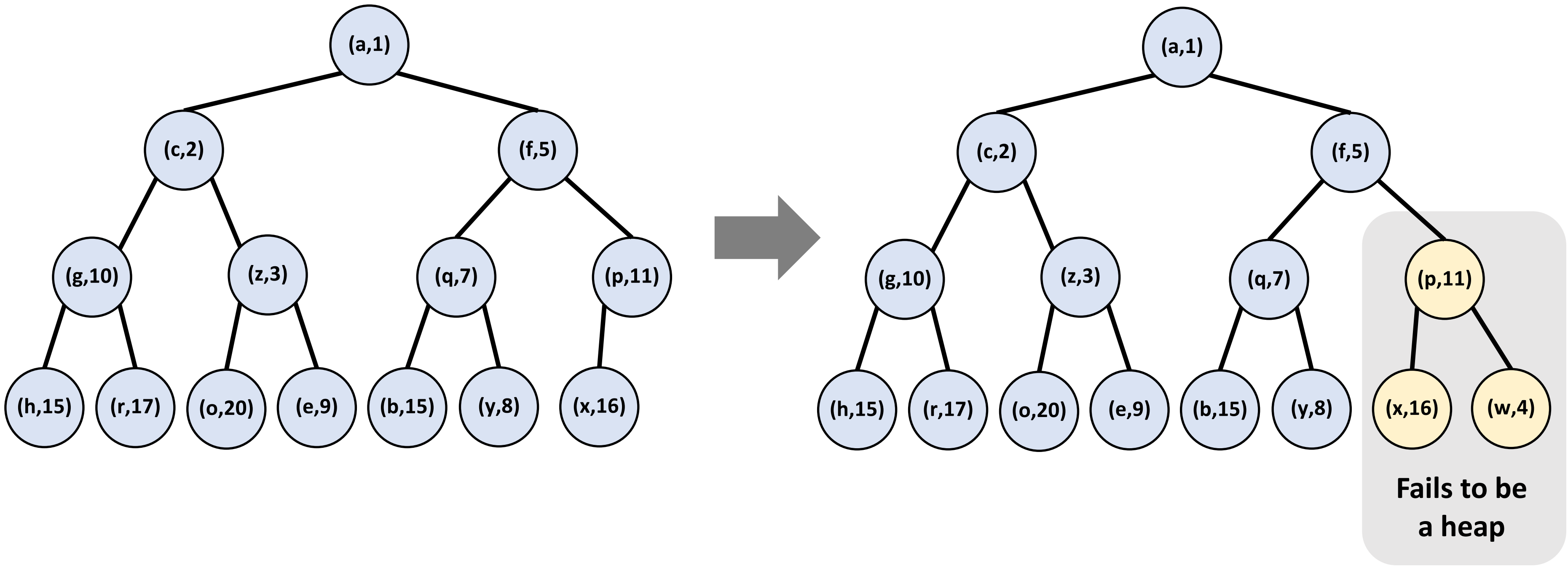


Ordering only along tree

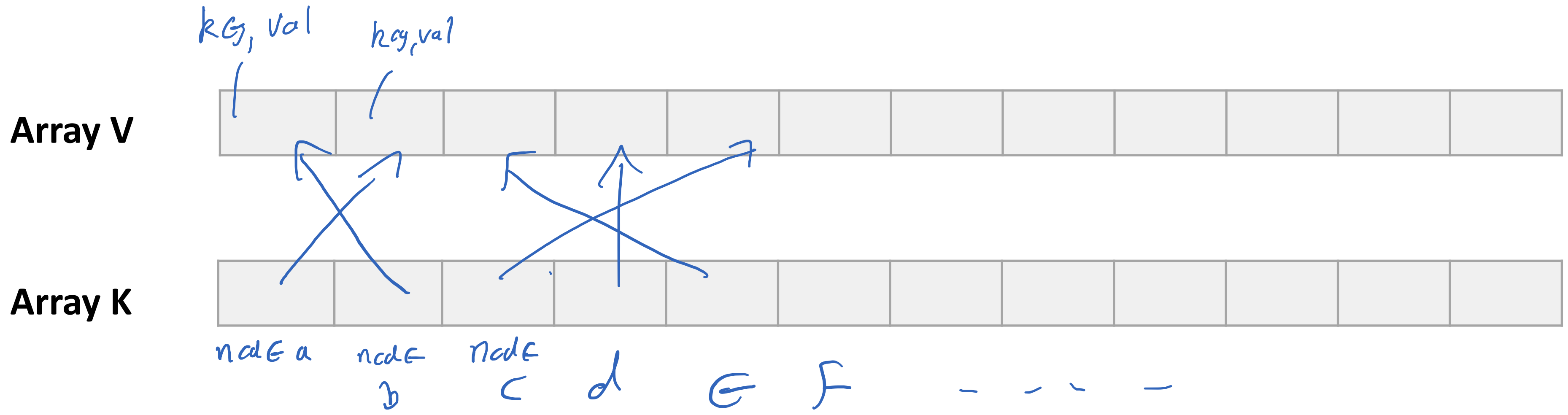
not across tree

Not Sorted

Implementing Insert



Implementation of Priority Queue Using Arrays



- Maintain an array V holding the (key,value) at each node the binary tree
- Maintain an array K mapping keys index
  - Can find the value for a given key in $O(1)$ time

Binary Heaps

- **Heapify:**
  - O(1) time to fix a single triple
  - With n keys, might have to fix O(log n) triples
  - Total time to heapify is O(log n)

- **Lookup** takes O(1) time
- **ExtractMin** takes O(log n) time
- **DecreaseKey** takes O(log n) time
- **Insert** takes O(log n) time

Implementing Dijkstra with Heaps

Lookup takes O(1) time
ExtractMin takes O(log n) time
DecreaseKey takes O(log n) time
Insert takes O(log n) time

```
Dijkstra(G = (V,E,{ℓ(e)}, s):
  Let Q be a new heap
  Let parent[u] ← ⊥ for every u
  Insert(Q,s,0), Insert(Q,u,∞) for every u != s

  While (Q is not empty):
    (u,d[u]) ← ExtractMin(Q)

    For ((u,v) in E):
      d[v] ← Lookup(Q,v)
      If (d[v] > d[u] + ℓ(u,v)):
        DecreaseKey(Q,v,d[u] + ℓ(u,v))
        parent[v] ← u

  Return (d, parent)
```

build heap

remove one item and heapify

lookup and modification

How much time does Dijkstra take?

n items

cost per item

$n \log n$

Repeat n times

log n

Repeated deg(u) times

$\log n \cdot \deg(u)$

\# edges

Total time: $\sum_v O(\log n) + O(\deg(u) \log n) = O((m+n)\log n)$

$= O(m \log n)$

Dijkstra Summary:

- **Dijkstra's Algorithm** solves **single-source shortest paths** in non-negatively weighted graphs
  - Algorithm can fail if edge weights are negative!

- **Implementation:**
  - A **priority queue** supports all necessary operations
  - Implement priority queues using **binary heaps**
  - Overall running time of Dijkstra: $O(m \log n)$

only paying
log n cost
due to weights

- **Compare to BFS**

Recurrence

- **Subproblems:** Let $\mathrm{OPT}(v, j)$ be the length of the shortest path from s to v with at most j hops
- **Case u:** (u, v) is final edge on the shortest j-hop $s \rightsquigarrow v$ path

was able to reach with fewer hops

Recurrence:

$$\mathrm{OPT}(v, j) = \min\left\{\mathrm{OPT}(v, j-1), \min_{(u,v)\in E}\{\mathrm{OPT}(u, j-1) + \ell_{u,v}\}\right\}$$

$\mathrm{OPT}(s, j) = 0$ for every j

$\mathrm{OPT}(v, 0) = \infty$ for every v

$\boxed{\text{Compute } \mathrm{OPT}(v, n)}$

vertices

Implementation (Bottom Up DP)

```
Shortest-Path(G, s)
    foreach node v ∈ V
        D[v,0] ← ∞
        P[v,0] ← ⊥
    D[s,0] ← 0

    for i = 1 to n-1
        foreach node v ∈ V
            D[v,i] ← D[v,i-1]
            P[v,i] ← P[v,i-1]
            foreach edge (u,v) ∈ E
                if (D[u,i-1] + ℓuv < D[v,i])
                    D[v,i] ← D[u,i-1] + ℓuv
                    P[v,i] ← u
```

} base cases

— no path has more than n-1 length

O(n) iterations

O(n) iterations

} copying data

— edge incoming to v

indegree(V)

nodes w/ an edge into V

$\sum_{i=1}^{n-1} \underbrace{\sum_{v \in V} \text{indegree}(v)}_{= m \text{ (\# edges)}}$

$O(nm)$

Running time: $O(nm)$

Space: $O(n^2)$