# Algorithms: Dynamic Programming (Optimal Binary Search Trees) and Graphs

Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 13, 2.04.2025

# DYNAMIC PROGRAMMING

## (An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

# Dynamic Programming (DP)

Main idea:

- Remember calculations already made
- Saves enormous amounts of computation

Allows to solve many optimization problems

- Always at least one question in google code jam needs DP

# Key elements in designing a DP-algorithm

## Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve and the optimal solution solves the subproblems optimally

## Overlapping subproblems

- A naive recursive algorithm may revisit the same (sub)problem over and over.
- Top-down with memoization

  Solve recursively but store each result in a table
- Bottom-up

  Sort the subproblems and solve the smaller ones first; that way, when solving a subproblem, have already solved the smaller subproblems we need

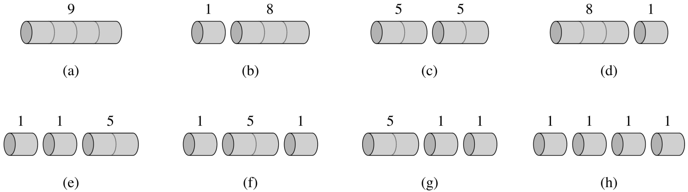
# ROD CUTTING

# Rod cutting

## Definition

**INPUT:** A length $n$ and table of prices $p_i$, for $i = 1, \ldots, n$

**OUTPUT:** The maximum revenue obtainable for rods whose lengths sum up to $n$, computed as the sum of the prices for the individual rods.



# Dynamic programming algorithm

Choice: where to make the leftmost cut

Optimal substructure: to obtain an optimal solution, we need to cut the remaining piece in an optimal way

Hence, if we let $r(n)$ be the optimal revenue from a rod of length $n$, we can express $r(n)$ recursively as follows

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \end{cases}$$

Optimal substructure: Solve recurrence using top-down with memoization or bottom-up which yields an algorithm that runs in time $\Theta(n^2)$.

| Parenthesization | Cost computation | Cost |
|---|---|---|
| $A \times ((B \times C) \times D)$ | $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$ | $120,200$ |
| $(A \times (B \times C)) \times D$ | $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$ | $60,200$ |
| $(A \times B) \times (C \times D)$ | $50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$ | $7,000$ |

# MATRIX-CHAIN MULTIPLICATION

# Matrix-chain multiplication

## Definition

INPUT: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of $n$ matrices, where for $i = 1, 2, \ldots, n$, matrix $A_i$ has dimension $p_{i-1} \times p_i$

OUTPUT: A full parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications

Example: Optimal parenthesization of $A_{4,3} \times B_{3,5} \times C_{5,2}$ is

$$(A_{4,3} \times (B_{3,5} \times C_{5,2}))$$

and requires $3 \cdot 5 \cdot 2 + 4 \cdot 3 \cdot 2$ multiplications.

# Dynamic programming algorithm

Choice: where to make the outermost parenthesis

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

Optimal substructure: to obtain an optimal solution, we need to parenthesize the two remaining expressions in an optimal way

Hence, if we let $m[i,j]$ be the optimal value for chain multiplication of matrices $A_i, \ldots, A_j$, we can express $m[i,j]$ recursively as follows

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise if } i < j \end{cases}$$

Overlapping subproblems: Solve recurrence using top-down with memoization or bottom-up which yields an algorithm that runs in time $\Theta(n^3)$.

# LONGEST COMMON SUBSEQUENCE

# Longest common subsequence

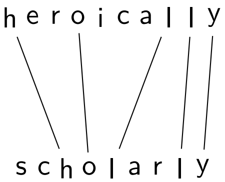
## Definition

INPUT: 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$.

OUTPUT: A subsequence common to both whose length is longest.

A subsequence doesn't have to be consecutive, but it has to be in order

Example:

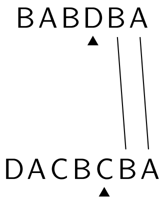


# Dynamic programming comes to the rescue

Start at the end of both words and move to the left step-by-step

Choice? If the same, pick letter to be in the subsequence

If not the same, optimal subsequence can be obtained by moving a step to the left in one of the words

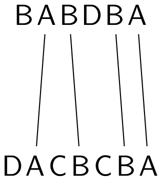


# Dynamic programming comes to the rescue

Start at the end of both words and move to the left step-by-step

Choice? If the same, pick letter to be in the subsequence

If not the same, optimal subsequence can be obtained by moving a step to the left in one of the words



# Dynamic programming algorithm

Let $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

Choice:

If $x_i = y_j$ then either

- OPT "matches" $x_i$ with $y_j$ and remaining OPT is in $(X_{i-1}, Y_{j-1})$;
- OPT is in $(X_{i-1}, Y_j)$; or
- OPT is in $(X_i, Y_{j-1})$

If $x_i \neq y_j$ then either

- OPT is in $(X_{i-1}, Y_j)$; or
- OPT is in $(X_i, Y_{j-1})$

We proved that we can assume that OPT "matches" $x_i$ with $y_j$ if they are equal so we can simplify the first case

# Recursive formulation

Define $c[i,j]$ = length of LCS of $X_i$ and $Y_j$. We want $c[m,n]$

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Naive implementation solves same problems many many times
- Solve with Bottom-Up or Top-Down with Memoization in time $O(m \cdot n)$.

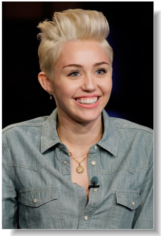
# Pseudocode and analysis

```
LCS-LENGTH(X, Y, m, n)
  let b[1..m, 1..n] and c[0..m, o..n] be new tables
  for i = 1 to m
      c[i, 0] = 0
  for j = 0 to n
      c[0, j] = 0
  for i = 1 to m
      for j = 1 to n
          if x_i == y_j
              c[i, j] = c[i − 1, j − 1] + 1
              b[i, j] = "↖"
          else if c[i − 1, j] ≥ c[i, j − 1]
                  c[i, j] = c[i − 1, j]
                  b[i, j] = "↑"
              else c[i, j] = c[i, j − 1]
                  b[i, j] = "←"
  return c and b
```

- Time dominated by instructions inside the two nested loops which execute $m \cdot n$ times
- Total time is $\Theta(m \cdot n)$.

# OPTIMAL BINARY SEARCH TREES

# Searching on Facebook



More popular than

# Optimal binary search trees

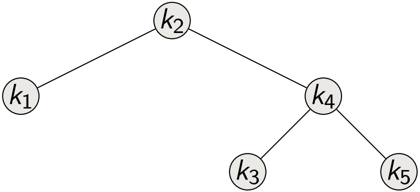
- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of $n$ distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).
- Want to build a binary search tree from the keys
- For $k_i$, have probability $p_i$ that a search is for $k_i$
- Want BST with minimum expected search cost
- Actual cost $= \#$ of items examined

  For key $k_i$, cost $= \text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ denotes the depth of $k_i$ in BST $T$

$$\begin{aligned}\mathbb{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) p_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i\end{aligned}$$

# Example

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | .25 | .2 | .05 | .2 | .3 |

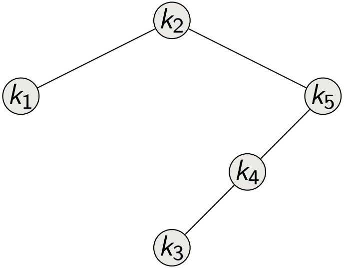



| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i) \cdot p_i$ |
|---|---|---|
| 1 | 1 | .25 |
| 2 | 0 | 0 |
| 3 | 2 | .1 |
| 4 | 1 | .2 |
| 5 | 2 | .6 |
| | | 1.15 |

Therefore, $\mathbb{E}[\text{search cost}] = 2.15$

# Example

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | .25 | .2 | .05 | .2 | .3 |




| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i) \cdot p_i$ |
|---|---|---|
| 1 | 1 | .25 |
| 2 | 0 | 0 |
| 3 | 3 | .15 |
| 4 | 2 | .4 |
| 5 | 1 | .3 |
| | | 1.10 |

Therefore, $\mathbb{E}[\text{search cost}] = 2.10$, which turns out to be optimal

# Observations

- Optimal BST might not have smallest height
- Optimal BST might not have highest-probability key at root

Build by exhaustive checking?

- Construct each $n$-node BST
- For each put in keys
- Then compute expected search cost
- But there are exponentially many trees

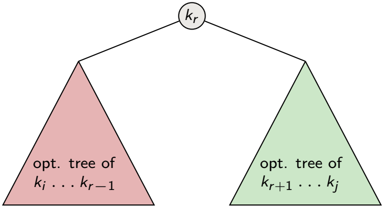
DP comes to the rescue :)

# Optimal substructure

A binary search tree can be built by first picking the root and then building the subtrees recursively

After picking root solution to subtrees must be optimal

Build tree of nodes $k_i < k_{i+1} < \cdots < k_{j-1} < k_j$ by selecting best root $r$:




$$\begin{aligned}\mathbb{E}[\text{search cost}] = p_r \\ + p_i + \cdots + p_{r-1} + \mathbb{E}[\text{search cost left subtree}] \\ + p_{r+1} + \cdots + p_j + \mathbb{E}[\text{search cost right subtree}]\end{aligned}$$

# Recursive formulation

- Let $e[i,j]$ = expected search cost of optimal BST of $k_i \dots k_j$

$$e[i,j] = \begin{cases} 0 & \text{if } i = j+1 \\ \min_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + \sum_{\ell=i}^{j} p_\ell \} & \text{if } i \le j \end{cases}$$

- Solve using bottom-up or top-down with memoization

# Bottom-up example

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | .25 | .2 | .05 | .2 | .3 |

$$e[i,j] = \begin{cases} 0 & \text{if } i = j+1 \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + \sum_{\ell=i}^{j} p_\ell\} & \text{if } i \le j \end{cases}$$

| e | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 1 | 0 | .25 | .65 | .8 | 1.25 | 2.1 |
| 2 | | 0 | .2 | .3 | .75 | 1.35 |
| 3 | | | 0 | .05 | .3 | .85 |
| 4 | | | | 0 | .2 | .7 |
| 5 | | | | | 0 | .3 |
| 6 | | | | | | 0 |

Optimal BST has expected search cost 2.1
Can save decisions to reconstruct tree

# Runtime Analysis

```
OPTIMAL-BST(p, q, n)
  let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
  for i = 1 to n + 1
      e[i, i − 1] = 0
      w[i, i − 1] = 0
  for l = 1 to n
      for i = 1 to n − l + 1
          j = i + l − 1
          e[i, j] = ∞
          w[i, j] = w[i, j − 1] + p_j
          for r = i to j
              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
              if t < e[i, j]
                  e[i, j] = t
                  root[i, j] = r
  return e and root
```

- Runtime dominated by three nestled loops: total time is $\Theta(n^3)$
- Alternatively, $\Theta(n^2)$ cells to fill in
  Most cells take $\Theta(n)$ time to fill in

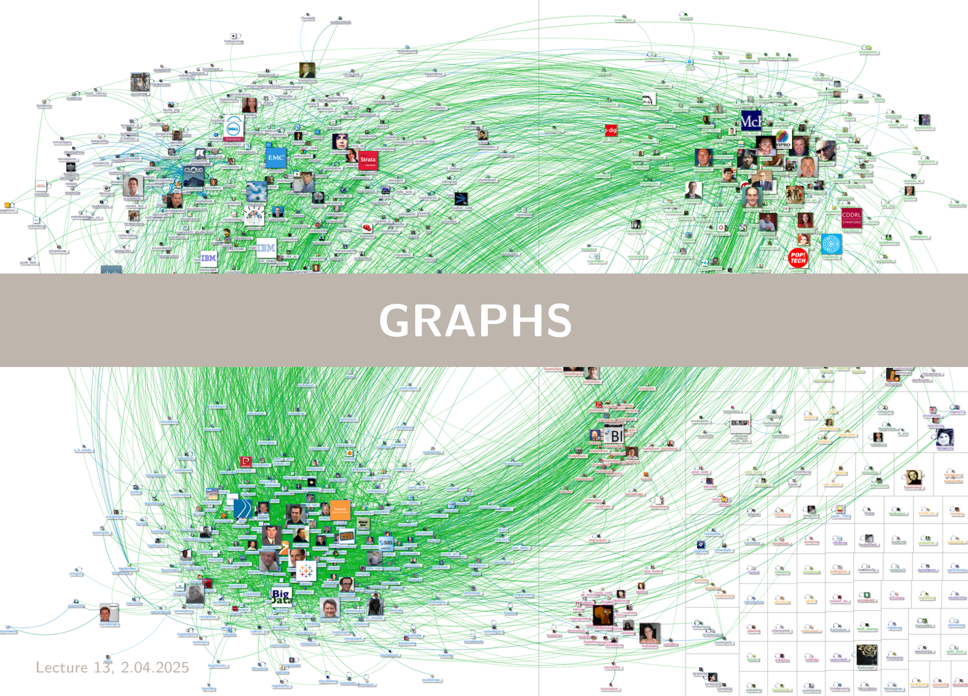
# Runtime Analysis

```
OPTIMAL-BST(p, q, n)
  let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
  for i = 1 to n + 1
      e[i, i − 1] = 0
      w[i, i − 1] = 0
  for l = 1 to n
      for i = 1 to n − l + 1
          j = i + l − 1
          e[i, j] = ∞
          w[i, j] = w[i, j − 1] + p_j
          for r = i to j
              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
              if t < e[i, j]
                  e[i, j] = t
                  root[i, j] = r
  return e and root
```

- Runtime dominated by three nestled loops: total time is $\Theta(n^3)$
- Alternatively, $\Theta(n^2)$ cells to fill in
  Most cells take $\Theta(n)$ time to fill in
  Hence, total time is $\Theta(n^3)$

# Summary of Dynamic Programming

- Identify choices and optimal substructure
- Write optimal solution recursively as a function of smaller subproblems
- Use top-down with memoization or bottom-up to solve the recursion efficiently (without repeatedly solving the same subproblems)
- Do a lot of exercises!



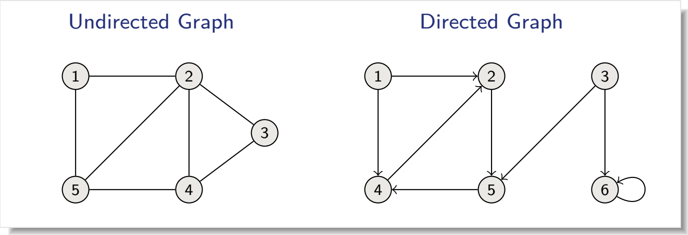
# GRAPHS

# Graphs

A graph $G = (V, E)$ consists of

- a vertex set $V$
- an edge set $E$ that contain (ordered) pairs of vertices

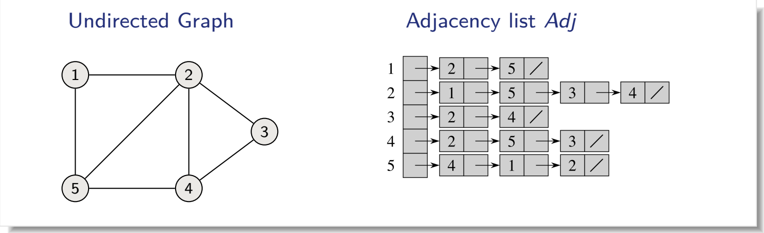
A graph can be undirected, directed, vertex-weighted, edge-weighted, etc.




How to represent a graph in the computer?

# Adjacency Lists

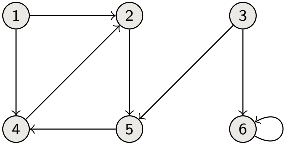
- Array *Adj* of $|V|$ lists, one per vertex
- Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$ (works for both undirected and directed graphs)



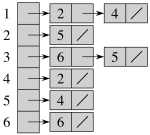
# Adjacency Lists

- Array *Adj* of $|V|$ lists, one per vertex
- Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$ (works for both undirected and directed graphs)
- In pseudocode, we will denote the array as attribute $G.Adj$, so we will see notation such as $G.Adj[u]$.

Directed Graph




Adjacency list *Adj*

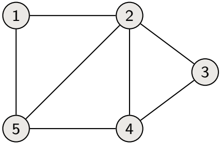


# Adjacency matrix

- A $|V| \times |V|$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 & \text{if}(i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Undirected Graph




Adjacency matrix

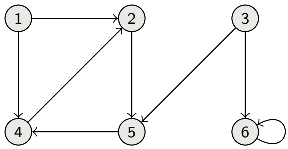
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

# Adjacency matrix

- A $|V| \times |V|$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 & \text{if}(i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Directed Graph




Adjacency matrix

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Comparison of adjacency list and adjacency matrix

**Adjacency list**

Space $= \Theta(V + E)$

Time: to list all vertices adjacent to $u$: $\Theta(\text{degree}(u))$

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$

**Adjacency matrix**

Space $= \Theta(V^2)$

Time: to list all vertices adjacent to $u$: $\Theta(V)$

Time: to determine whether $(u, v) \in E$: $\Theta(1)$

---

We can extend both representations to include other attributes such as edge weights

# TRAVERSING/SEARCHING A GRAPH

# Breadth-First Search

## Definition

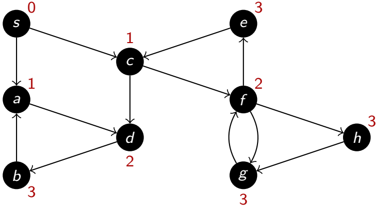
INPUT: Graph $G = (V, E)$, either directed or undirected and source vertex $s \in V$

OUTPUT: $v.d =$ distance (smallest number of edges) from $s$ to $v$, for all $v \in V$

Idea:

- Send a wave out from $s$
- First hits all vertices 1 edge from $s$
- From there, hits all vertices 2 edges from $s$ ...

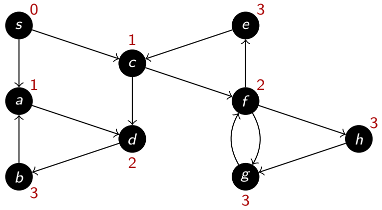
# Example of Breadth-first search




Queue Q = nil

# Pseudocode of Breadth-first search

BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)




Queue Q = nil

# Analysis

Informal Idea of correctness (formal proof in book):

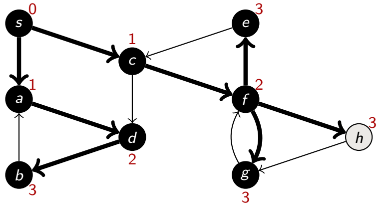
- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

Runtime analysis: O(V+E)

- $O(V)$ because each vertex enqueued at most once
- $O(E)$ because every vertex dequeued at most once and we examine $(u, v)$ only when $u$ is dequeued. Therefore, every edge examined at most once if directed and at most twice if undirected

# Final notes on BFS

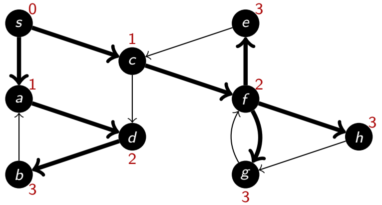
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = h

# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = nil

# Depth-First Search

## Definition

INPUT: Graph $G = (V, E)$, either directed or undirected

OUTPUT: 2 timestamps on each vertex: $v.d$ = **discovery time** and $v.f$ = **finishing time**

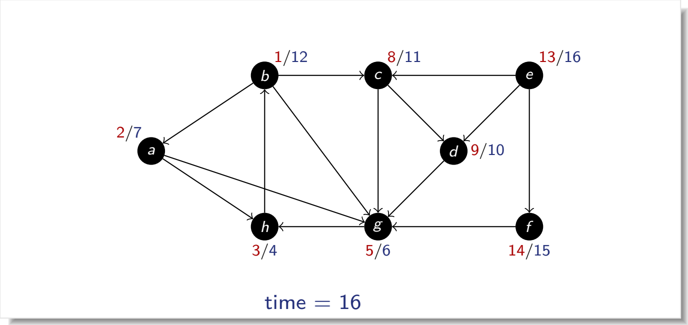
Idea:

- Methodically explore *every* edge
- Start over from different vertices as necessary
- As soon as we discover a vertex explore from it,
  - Unlike BFS, which explores vertices that are close to a source first

# Example of DFS

As DFS progresses, every vertex has a color:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



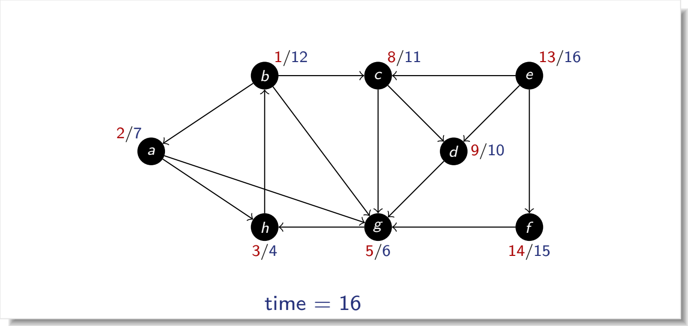
# Pseudocode of DFS

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```

# Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



# Analysis

DFS forms a depth-first forest comprised of $\geq 1$ depth-first trees. Each tree is made of edges $(u, v)$ such that $u$ is gray and $v$ is white when $(u, v)$ is explored.

Runtime analysis: $\Theta(V + E)$

- $\Theta(V)$ because each vertex is discovered once
- $\Theta(E)$ because each edge is examined once if directed graph and twice if undirected graph.

# Classification of edges

Tree edge: In the depth-first forest, found by exploring $(u, v)$

Back edge: $(u, v)$ where $u$ is a descendant of $v$

Forward edge: $(u, v)$ where $v$ is a descendant of $u$, but not a tree edge

Cross edge: any other edge

In DFS of an undirected graph we get only tree and back edges, no forward or cross-edges. Why?