

Greedy Algorithm

Ajit Kumar Behera

Greedy Algorithm

- Applicable for optimization problems
- Based on trying best current (local) choice
- Approach
 - At each step of algorithm
 - Choose best local solution
 - Hope local optimum lead to global solution
 - Not always, but in many situation, it works
- More efficient than dynamic programming.
- Does not guarantee an optimal solution always
- To use DP, show that the **principle of optimality applies** to the problem

Principle of optimality applies

Definition:

An optimal sequence of decision has the property that what ever the initial state and the decisions are, the remaining decisions must constitute an optimal decision with regard to the state resulting from the first decision.

Greedy Algorithm

- ❑ A problem has **n** inputs and require us to obtain a **subset** that satisfies some conditions.
- ❑ Any subset that satisfies these constraints is called **feasible solution**.
- ❑ We need to find a solution that either **maximize** or **minimize** a given **objective function**.
- ❑ A feasible solution, that does this is called an **optimal solution**.

Greedy Algorithm

Approach:

- Greedy method suggests that one can device an algorithm that works in **stages**, considering one input at a time.
- At each stage, a **decision** is made regarding whether a particular input is an **optimal solution**.

Greedy Algorithm

Example:

- Knapsack problem
- Activity selection problem
- Hoffman code
- Tree vertex splitting
- Job sequencing with deadlines
- Minimum spanning tree
- Single source shortest path

Knapsack problem

- Given a knapsack with weight $W > 0$.
- A set S of n items with weights $w_i > 0$ and profits $p_i > 0$ for $i = 1, \dots, n$.
- $S = \{ (\text{item}_1, w_1, p_1), (\text{item}_2, w_2, p_2), \dots, (\text{item}_n, w_n, p_n) \}$
- Find a subset of the items which does not exceed the weight W of the knapsack and maximizes the benefit.

Knapsack problem

Two types:

- **Fractional knapsack problem**
 - Applies to items s.a. bread, gold dust, sugar
- **0/1 knapsack problem(no fraction)**
 - 1 green shirt
 - 10 gold bricks
 - 5 pairs of socks

Knapsack problem

- We are given n objects and a knapsack or bag.
- Object i has a weight w_i and the knapsack has a capacity m .
- If a fraction x_i , $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m$$

$$\text{and } 0 \leq x_i \leq 1, \quad 0 \leq x_i \leq 1$$

Brute force!

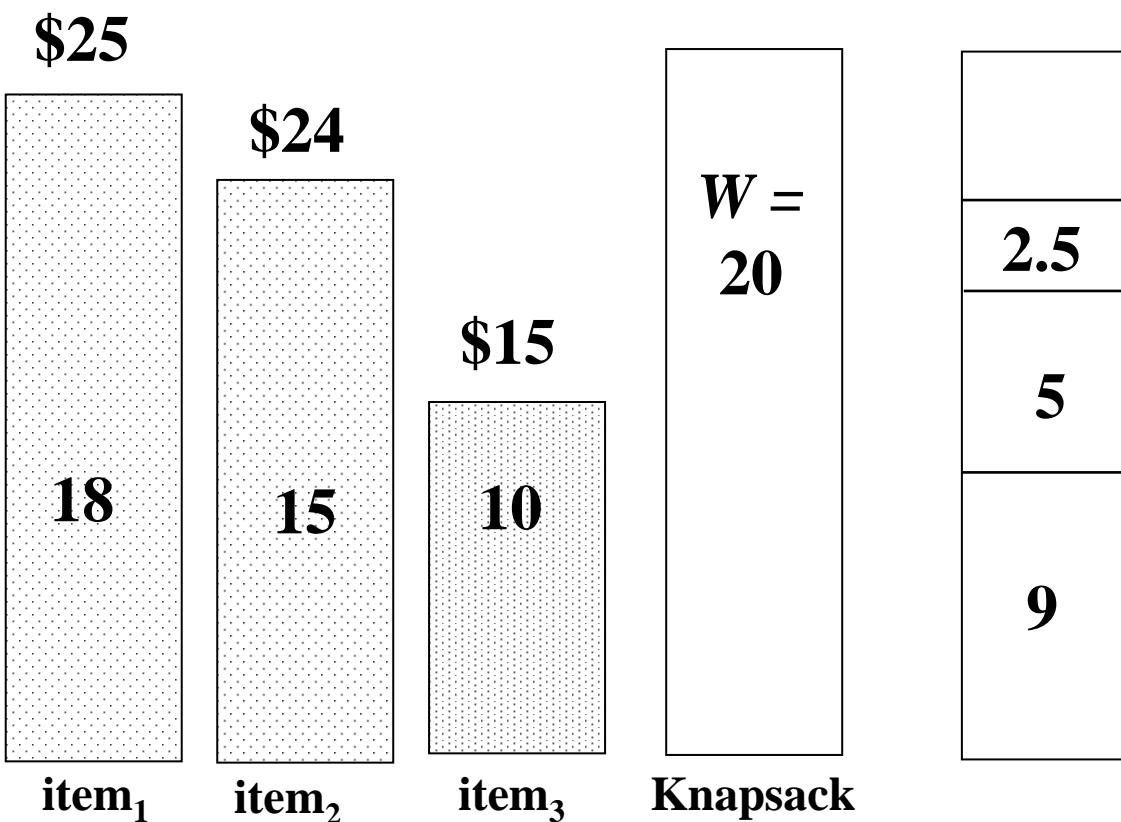
- Generate all 2^n subsets
- Discard all subsets whose sum of the weights exceed W (*not feasible*)
- Select the maximum total benefit of the remaining (feasible) subsets
- What is the run time?
 $O(n 2^n)$, $\Omega(2^n)$
- Lets try the obvious greedy strategy .

Example :

Greedy 1: Selection criteria: *select each item*

$$S = \{ (x_1, 18, \$25), (x_2, 15, \$24), (x_3, 10, \$15) \}, W=20, n=3$$

$$(x_1 \ x_2 \ x_3) = (1/2, 1/3, 1/4)$$



$$\sum w_i x_i = 16.5$$

$$\sum p_i x_i = 24.25$$

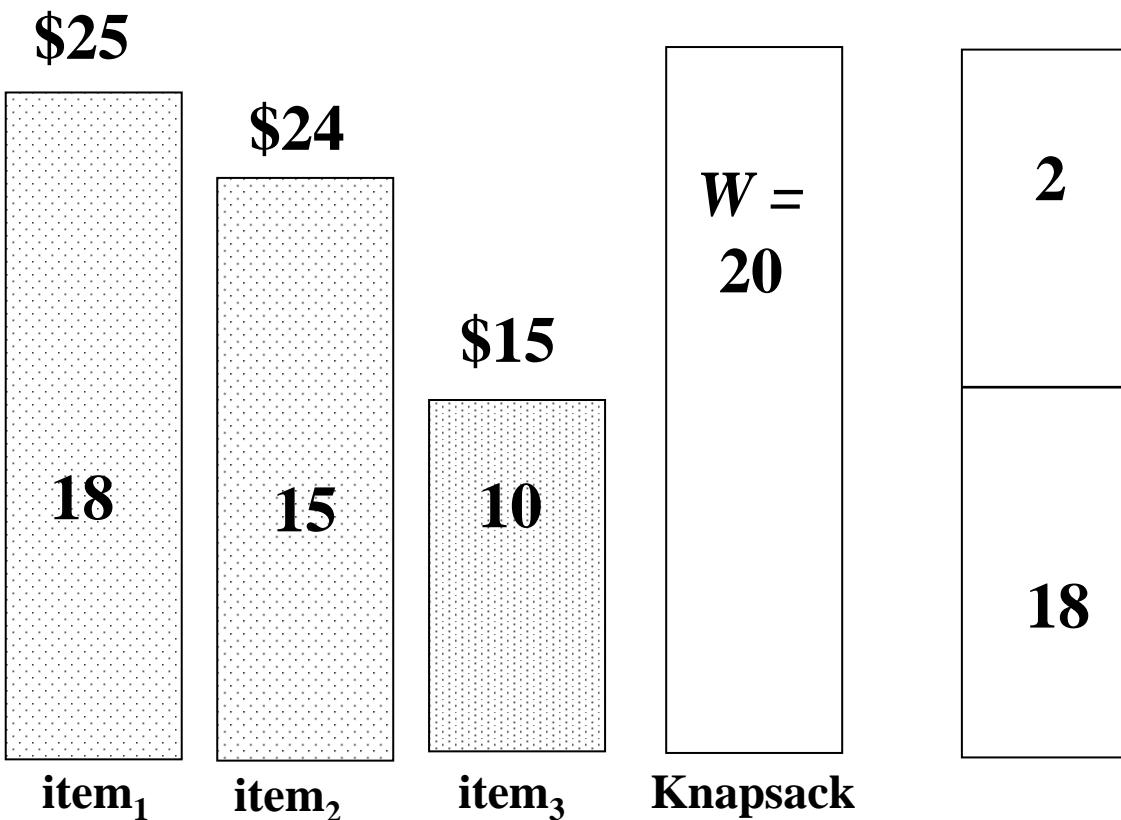
Feasible solution

Example :

Greedy 1: Selection criteria: *select maximum profitable item*

$$S = \{ (x_1, 18, \$25), (x_2, 15, \$24), (x_3, 10, \$15) \}, W=20, n=3$$

$$(x_1 \ x_2 \ x_3) = (1, 2/15, 0)$$



$$\sum w_i x_i = 20$$

$$\sum p_i x_i = 28.2$$

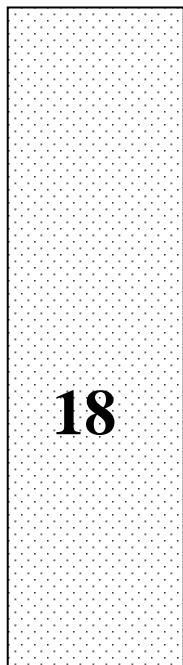
Feasible solution

Greedy 2: Selection criteria: *select minimum weight item*

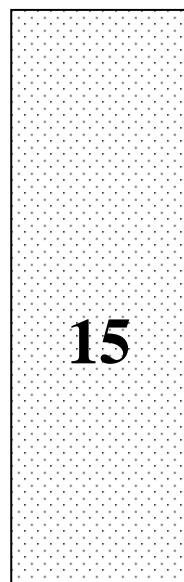
$$S = \{ (x_1, 18, \$25), (x_2, 15, \$24), (x_3, 10, \$15) \}, W=20, n=3$$

$$(x_1 \ x_2 \ x_3) = (0, 2/3, 1)$$

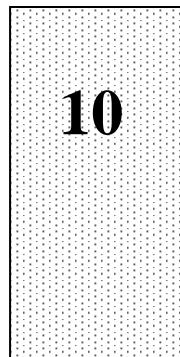
\$25



\$24



\$15



$$W = 20$$

Knapsack

10

10

$$\sum w_i x_i = 20$$

$$\sum p_i x_i = 31$$

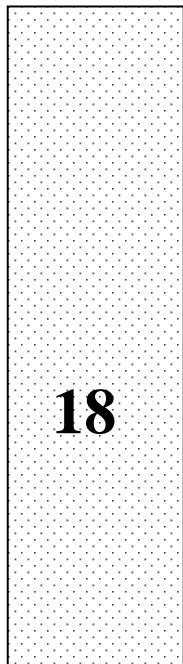
Feasible solution

Greedy 3: Selection criteria: *select maximum profit per unit of capacity*

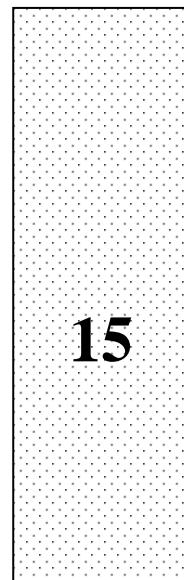
$$S = \{ (x_1, 18, \$25), (x_2, 15, \$24), (x_3, 10, \$15) \}, W=20, n=3$$

$$(x_1 \ x_2 \ x_3) = (0, 1, 1/2)$$

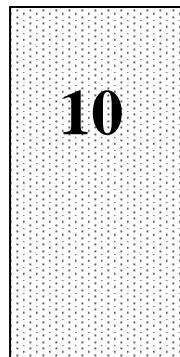
\$25



\$24



\$15



item₁

item₂

item₃

Knapsack

$$\sum w_i x_i = 20$$

$$\sum p_i x_i = 31.5$$

Feasible solution

Example with “brute force”

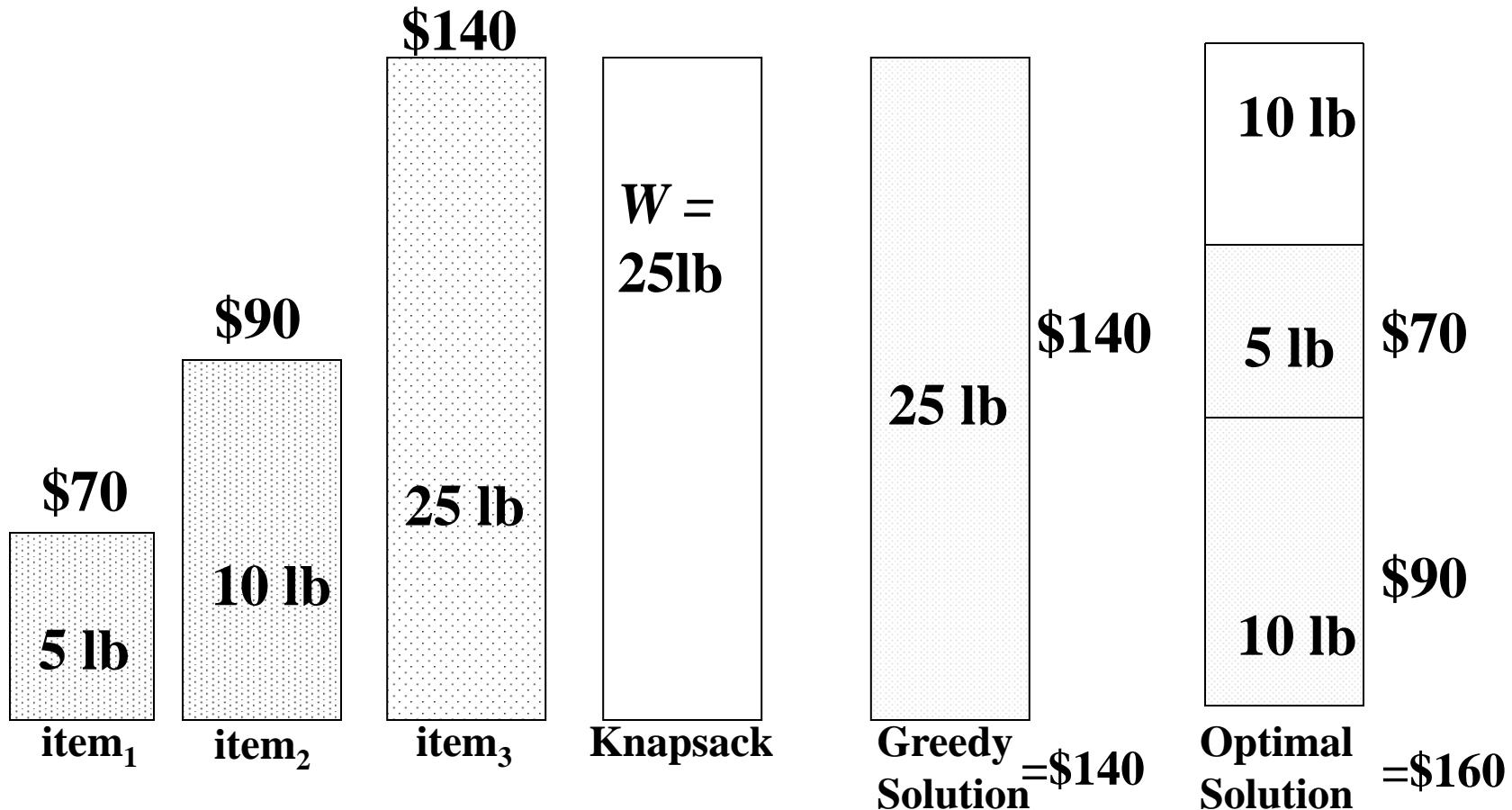
$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}$, $W=25$

Subsets:

1. {}
2. { ($item_1$, 5, \$70) } Profit=\$70
3. { ($item_2$,10, \$90) } Profit=\$90
4. { ($item_3$, 25, \$140) } Profit=\$140
5. { ($item_1$, 5, \$70), ($item_2$,10, \$90) }. Profit=\$160 ****
6. { ($item_2$,10, \$90), ($item_3$, 25, \$140) } exceeds W
7. { ($item_1$, 5, \$70), ($item_3$, 25, \$140) } exceeds W
8. { ($item_1$, 5, \$70), ($item_2$,10, \$90), ($item_3$, 25, \$140) }
exceeds W

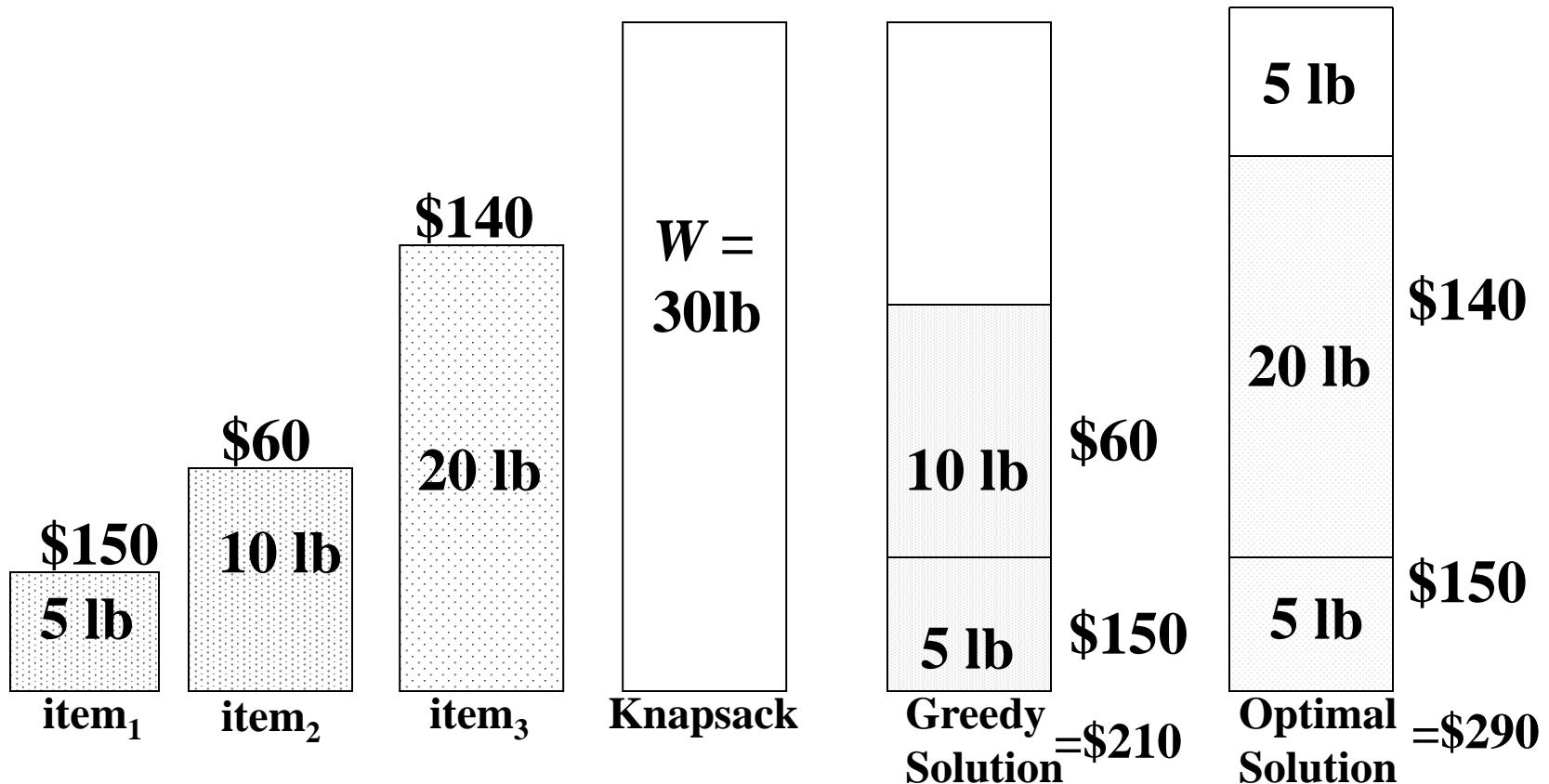
Greedy 1: Selection criteria: *Maximum beneficial item.*

$$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}$$



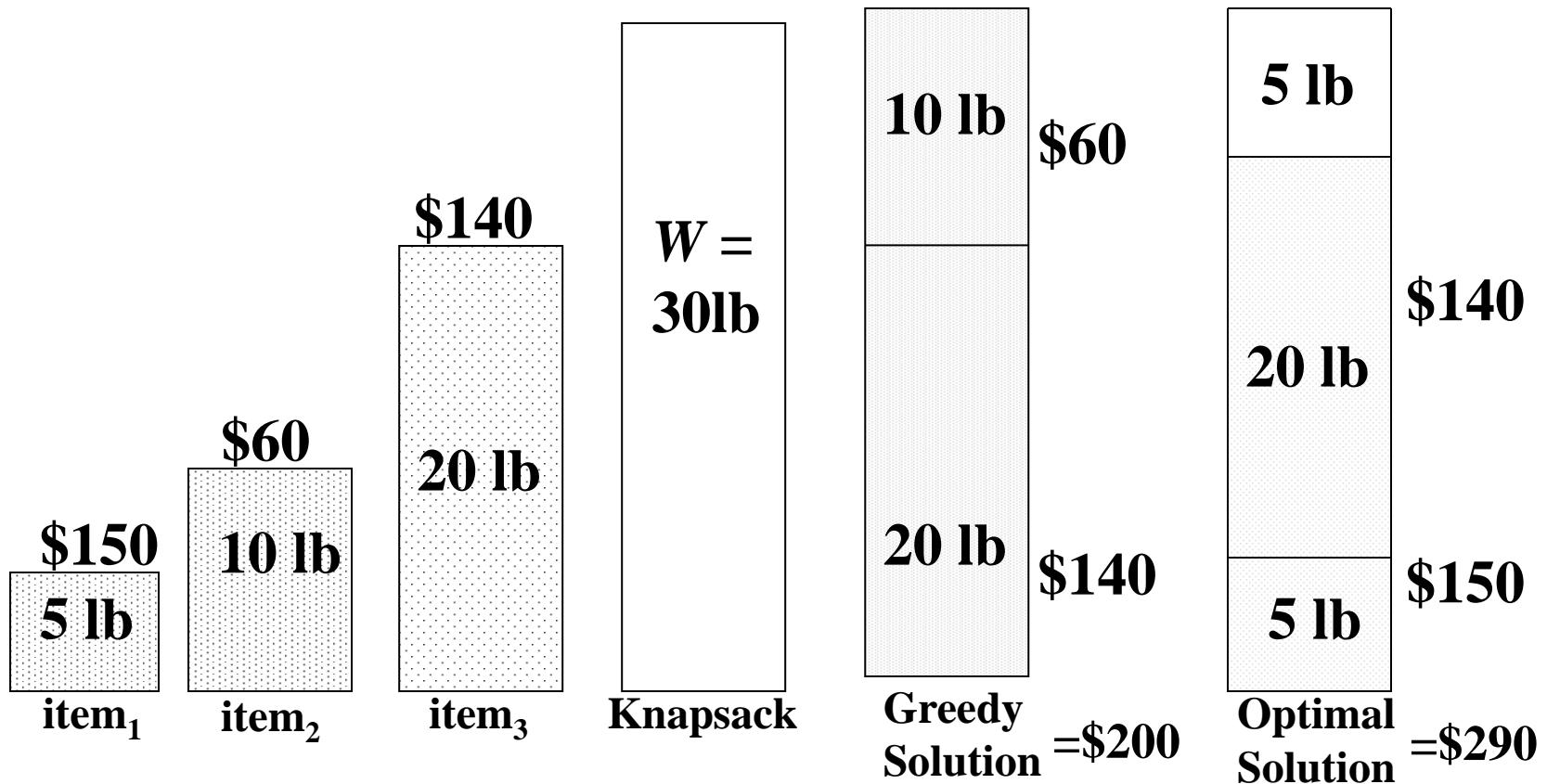
Greedy 2: Selection criteria: Minimum weight item

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



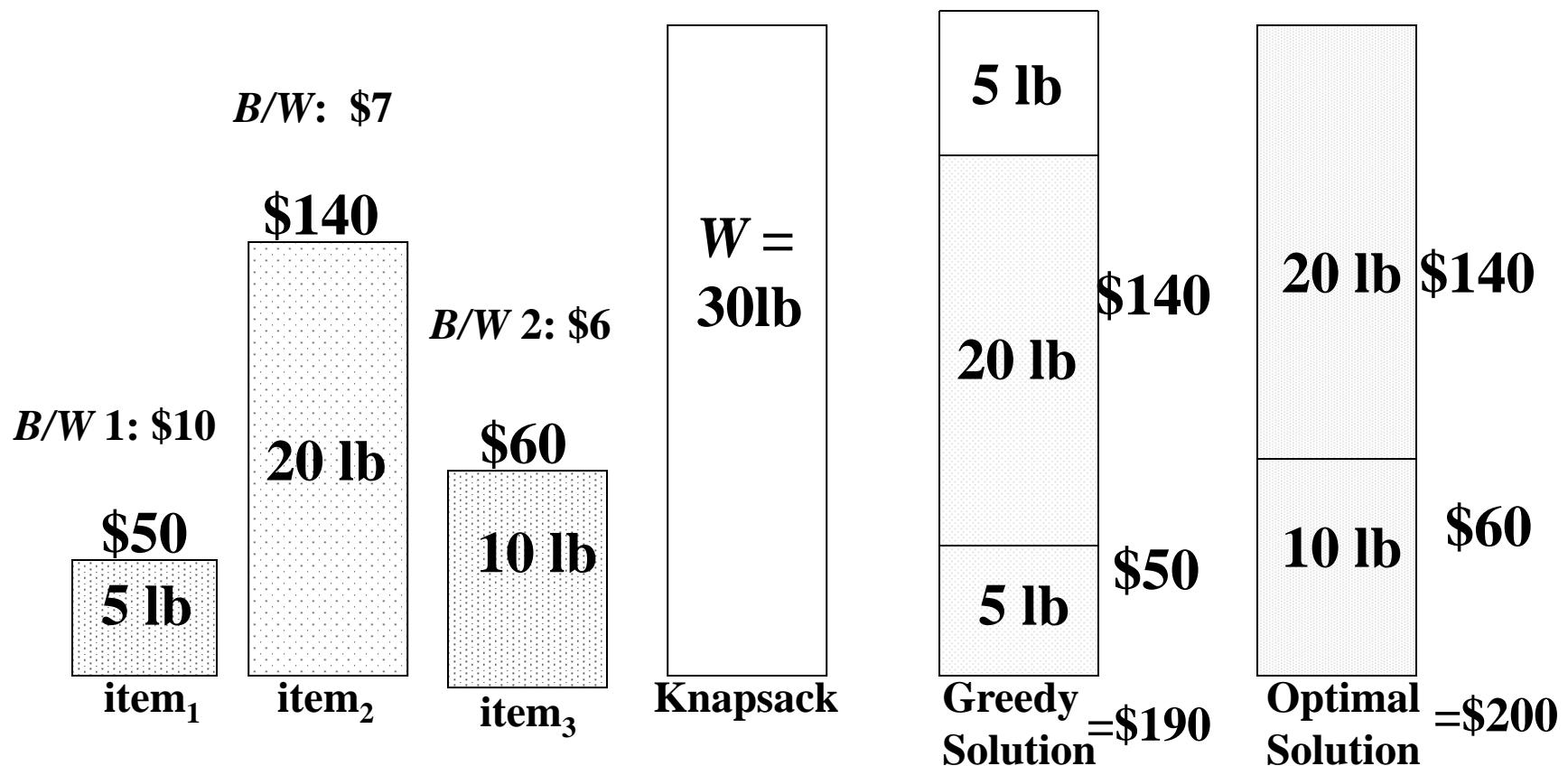
Greedy 3: Selection criteria: Maximum weight item

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



Greedy 4: Selection criteria: Maximum benefit per unit item

$$S = \{ (item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60), \}$$



Algorithm

GreedyKnapsack(m,n)

// $p[1..n]$ and $w[1..n]$ contain the profits and weights of n //objects ordered such that $\frac{p[i]}{w[i]} \geq \frac{p[i+1]}{w[i+1]}$.

// m is the knapsack size and $x[1..n]$ is the solution vector

{

 for $i = 1$ to n

$x[i] = 0$

$U = m$

 for $i = 1$ to n

 {

 if ($w[i] > U$) then break

$x[i] = 1$

$U = U - w[i]$

 }

 if ($i \leq n$) then $x[i] = \frac{U}{w[i]}$

}

Theorem

If $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$, then greedy knapsack generate an optimal solution.

An activity selection problem

Ajit Kumar Behera

An activity selection problem

Sl. No.	Activity	Start time	Finish time
1	DBMS	8:00	9:15
2	DAA	8:30	10:30
3	ACA	9:20	11:00
4	OS	10:00	01:30
5	SE	11:20	12:30

Class room no: 432

An activity selection problem

Problem:

- We have only one **resource**.
- The single resource need to be scheduled among several competing activities/processes such that the **resource utilization** can be **maximized**.

Claim: Select a maximum size set of mutually compatible activities.

- You can not be allowed to do two activities at the same time.

Formal definition

- Let $S = \{a_1, a_2, \dots, a_n\}$ be a set of activities which is to use a **resource** which can be used by only one activity at a time.
- Each activity a_i has a start time s_i and a finish time f_i , where $0 \leq s_i < f_i < \infty$
- If selected, activity a_i takes place during $[s_i, f_i)$
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap

Problem: select a maximum subset of mutually compatible activities

The Activity Selection Problem

Example: Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- What is the maximum number of activities that can be completed?
 - $\{a_3, a_9, a_{11}\}$ can be completed
 - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
 - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

Solution

we can solve the activity selection problem in several steps:

- Formulating a dynamic programming solution
- Develop a recursive greedy algorithm
- Converting the recursive algorithm to an iterative one.

The optimal substructure of the activity selection problem

Let S_{ij} is a subset defined as

$$S_{ij} = \{a_k \in S \mid f_i \leq s_k < f_k \leq s_j\}$$

- S_{ij} is the subset of activities in S that can start after the activity a_i finishes and finish before activity a_j starts.
- In fact, S_{ij} consists of all activities that are compatible with a_i and a_j and are also compatible with all activities that finish before a_i finishes and all activities that start after a_j
- For the entire problem, let two extra activities a_0 and a_{n+1} , where $f_0 = 0$ and $s_{n+1} = \infty$

The optimal substructure of the activity selection problem

Let us assume that the activities are sorted in increasing order of finish time. $f_0 \leq f_1 \leq \dots \leq f_n < f_{n+1}$

- So, $S_{ij} = \emptyset$, where $i \geq j$
- Our space of subproblem is to select a maximum size subset of mutually compatible activities from S_{ij} , for $0 \leq i < j \leq n + 1$

Find a substructure

Let S_{ij} is a non empty subproblem which include some activity a_k so that $f_i \leq s_k < f_k \leq s_j$

- It creates two subproblems S_{ik} and S_{kj} such that $S_{ij} = S_{ik} \cup S_{kj} \cup \{a_k\}$
- Our solution to S_{ij} is the union of the solution to S_{ik} and S_{kj} along with the activity a_k

We Show this is a Greedy Problem

Theorem 16.1

Consider any nonempty subproblem S_{ij} , and let a_m be the activity in S_{ij} with the earliest finish time:

$$f_m = \min \{f_k : a_k \in S_{ij}\} .$$

Then

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} .
2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

What are the consequences?

- Normally we have to inspect all subproblems, here we only have to choose one subproblem
- What this theorem says is that we only have to find the first subproblem with the smallest finishing time

An Iterative Approach

- We let f_i be the maximum finishing time for any activity in A

$$f_i = \max \{f_k : a_k \in A\}$$

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7           $i \leftarrow m$ 
8  return  $A$ 
```

- The overall complexity of this algorithm is $\Theta(n)$

Huffman Codes

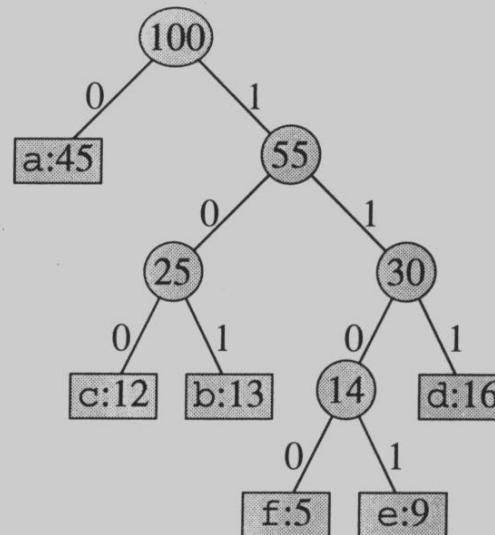
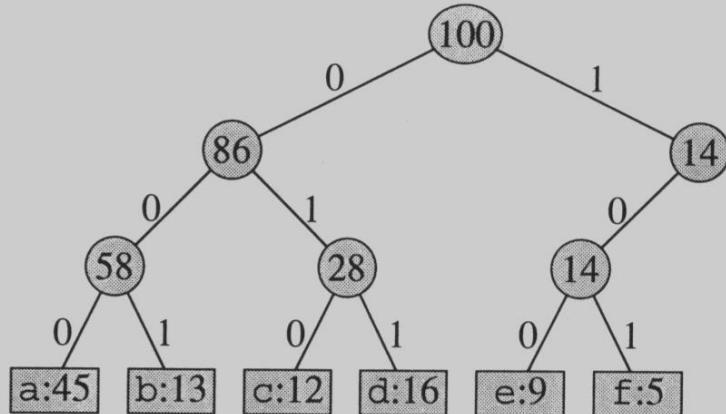
- Most character code systems (ASCII, unicode) use fixed length encoding
- If frequency data is available and there is a wide variety of frequencies, variable length encoding can save 20% to 90% space
- Which characters should we assign shorter codes; which characters will have longer codes?

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- At first it is not obvious how decoding will happen, but this is possible if we use prefix codes

Prefix Codes

- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode t as 01 and x as 01101 since 01 is a prefix of 01101
- By using a binary tree representation we will generate prefix codes provided all letters are leaves



Some Properties

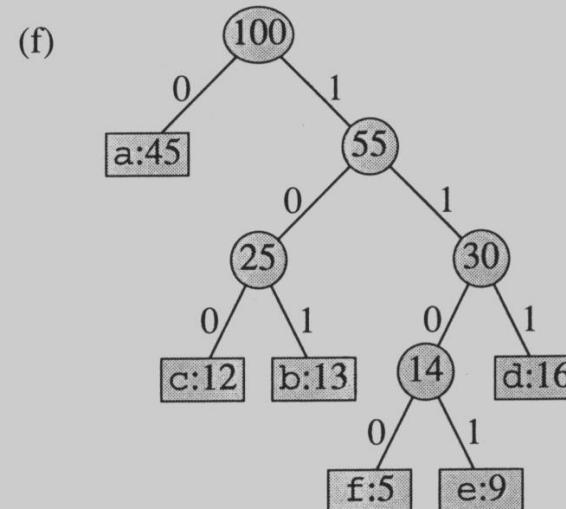
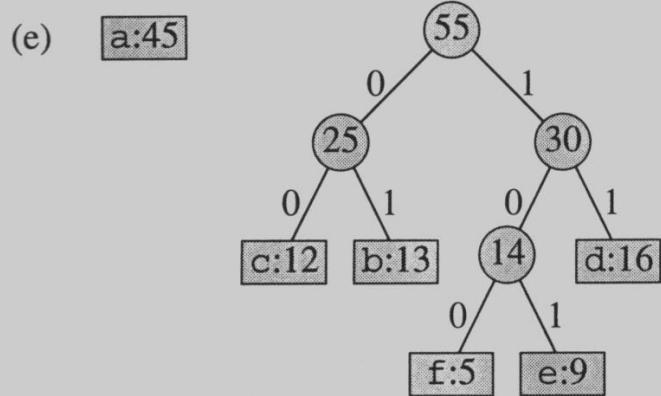
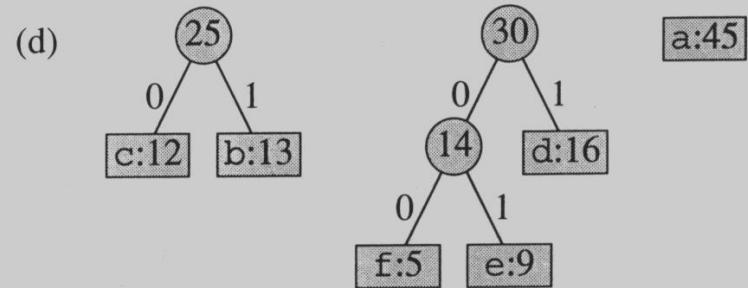
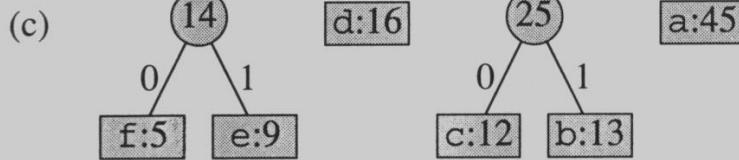
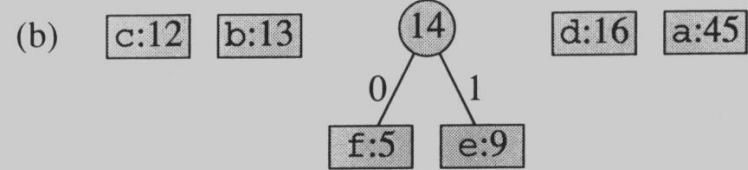
- Prefix codes allow easy decoding
 - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
 - Decode 001011101 going left to right, 0|01011101, a|0|1011101, a|a|101|1101, a|a|b|1101, a|a|b|e
- An optimal code must be a full binary tree (a tree where every internal node has two children)
- For C leaves there are C-1 internal nodes
- The number of bits to encode a file is

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

where $f(c)$ is the freq of c , $d_T(c)$ is the tree depth of c , which corresponds to the code length of c

Building the Encoding Tree

(a) f:5 e:9 c:12 b:13 d:16 a:45 .



HUFFMAN(C)

```
1  $n \leftarrow |C|$ 
2  $Q \leftarrow C$ 
3 for  $i \leftarrow 1$  to  $n - 1$ 
4     do allocate a new node  $z$ 
5          $left[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$ 
6          $right[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$ 
7          $f[z] \leftarrow f[x] + f[y]$ 
8          $\text{INSERT}(Q, z)$ 
9 return  $\text{EXTRACT-MIN}(Q)$            ▷ Return the root of the tree.
```

The Algorithm

- An appropriate data structure is a binary min-heap
- Rebuilding the heap is $\lg n$ and $n-1$ extractions are made, so the complexity is $O(n \lg n)$
- The encoding is NOT unique, other encoding may work just as well, but none will work better

Correctness of Huffman's Algorithm

- The following results are presented without proof

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Lemma 16.3

Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x, y removed and (new) character z added, so that $C' = C - \{x, y\} \cup \{z\}$; define f for C' as for C , except that $f[z] = f[x] + f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C' . Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

8 QUEENS PROBLEM USING BACK TRACKING

BACK TRACKING

- ✓ **Backtracking** is a general algorithm for finding all (or some) solutions to some computational problem, that *incrementally builds candidates to the solutions*, and abandons each partial candidate ‘ c ’ ("backtracks") as soon as it determines that ‘ c ’ cannot possibly be completed to a valid solution.
- ✓ Backtracking is an important tool for solving constraint satisfaction problems, such as *crosswords, verbal arithmetic, Sudoku, and many other puzzles*.

- ✓ It is also the basis of the so-called logic programming languages such as *Planner* and *Prolog*.
- ✓ The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s.
- ✓ The pioneer string-processing language SNOBOL (1962) may have been the first to provide a built-in general backtracking facility.

- ✓ The good example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight queens on a standard chessboard so that no queen attacks any other.
- ✓ In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns.
- ✓ Any partial solution that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution

WHAT IS 8 QUEEN PROBLEM?

- ✓ The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other.
- ✓ Thus, a solution requires that no two queens share the same row, column, or diagonal.
- ✓ The eight queens puzzle is an example of the more general **n -queens problem** of placing n queens on an $n \times n$ chessboard, where solutions exist for all natural numbers n with the exception of 1, 2 and 3.
- ✓ The solution possibilities are discovered only up to *23 queen*.

PROBLEM INVENTOR

- ✓ The puzzle was originally proposed in 1848 by the chess player **Max Bezzel**, and over the years, many mathematicians, including Gauss, have worked on this puzzle and its generalized n-queens problem.

SOLUTION INVENTOR

- ✓ The first solution for 8 queens were provided by ***Franz Nauck*** in 1850. Nauck also extended the puzzle to n-queens problem (on an $n \times n$ board—a chessboard of arbitrary size).
- ✓ In 1874, S. ***Günther*** proposed a method of finding solutions by using determinants, and ***J.W.L. Glaisher*** refined this approach.
- ✓ ***Edsger Dijkstra*** used this problem in 1972 to illustrate the power of what he called structured programming.
- ✓ He published a highly detailed description of the development of a ***depth-first backtracking algorithm***.

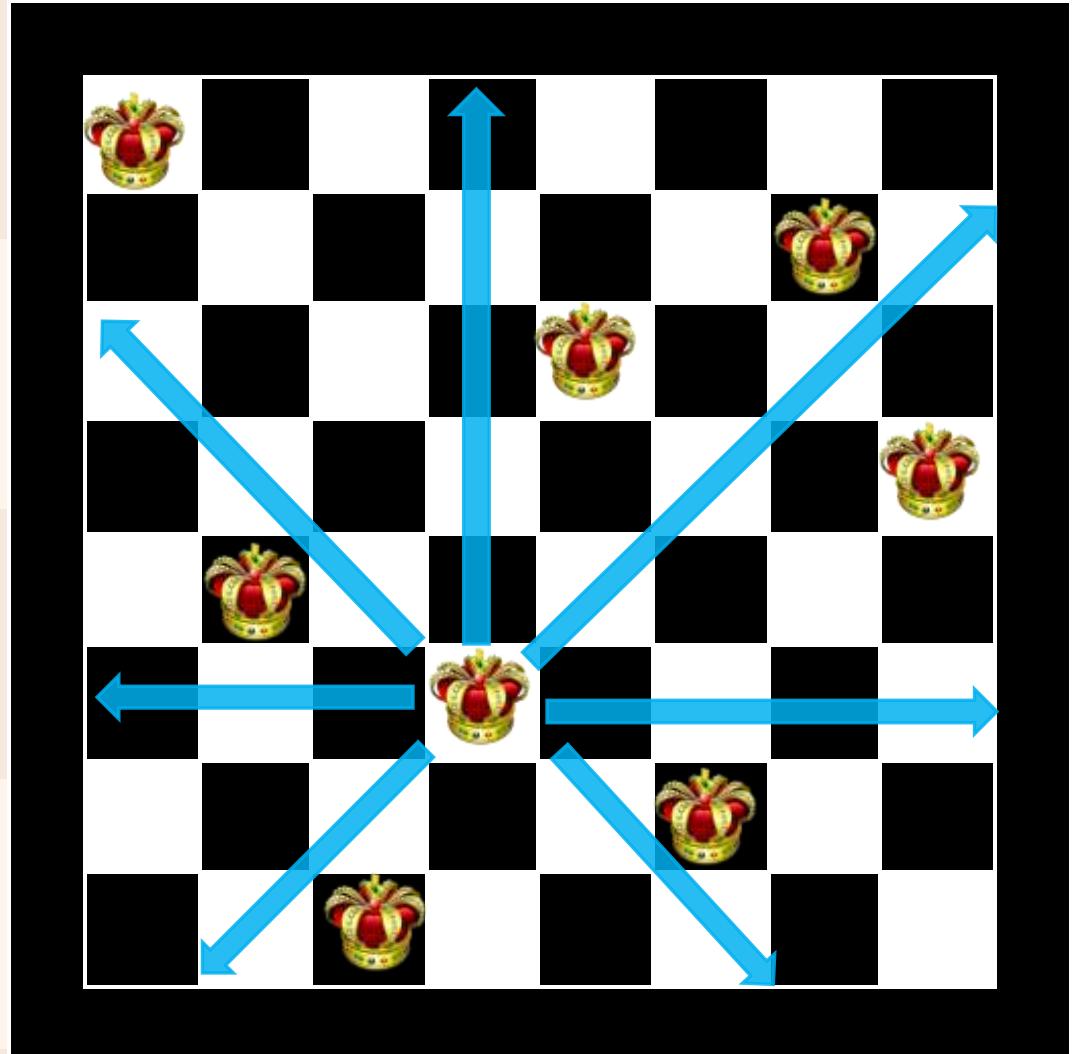
Formulation :

States: any arrangement of 0 to 8 queens on the board

Initial state: 0 queens on the board

Successor function: add a queen in any square

Goal test: 8 queens on the board, none attacked



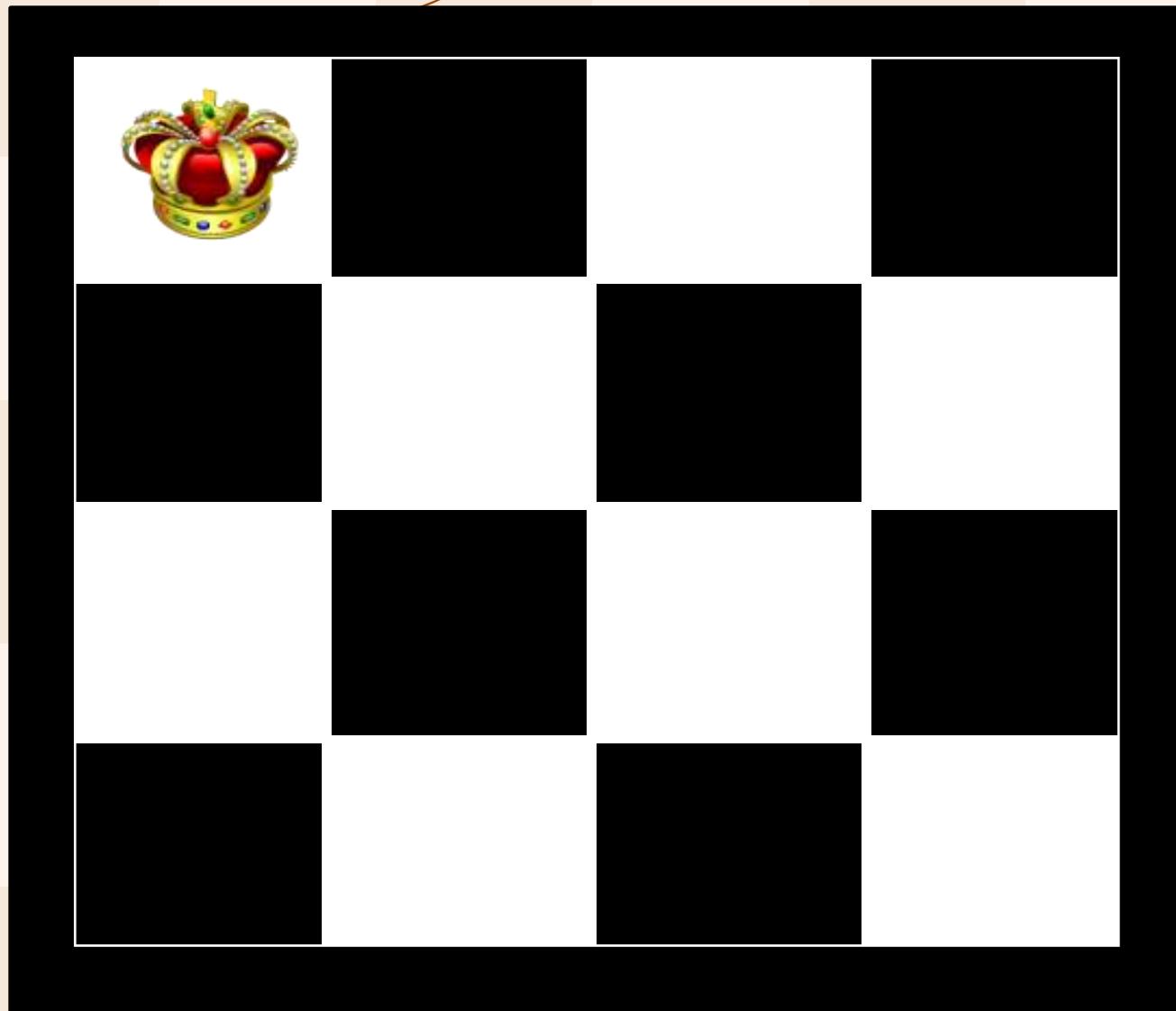
BACKTRACKING CONCEPT

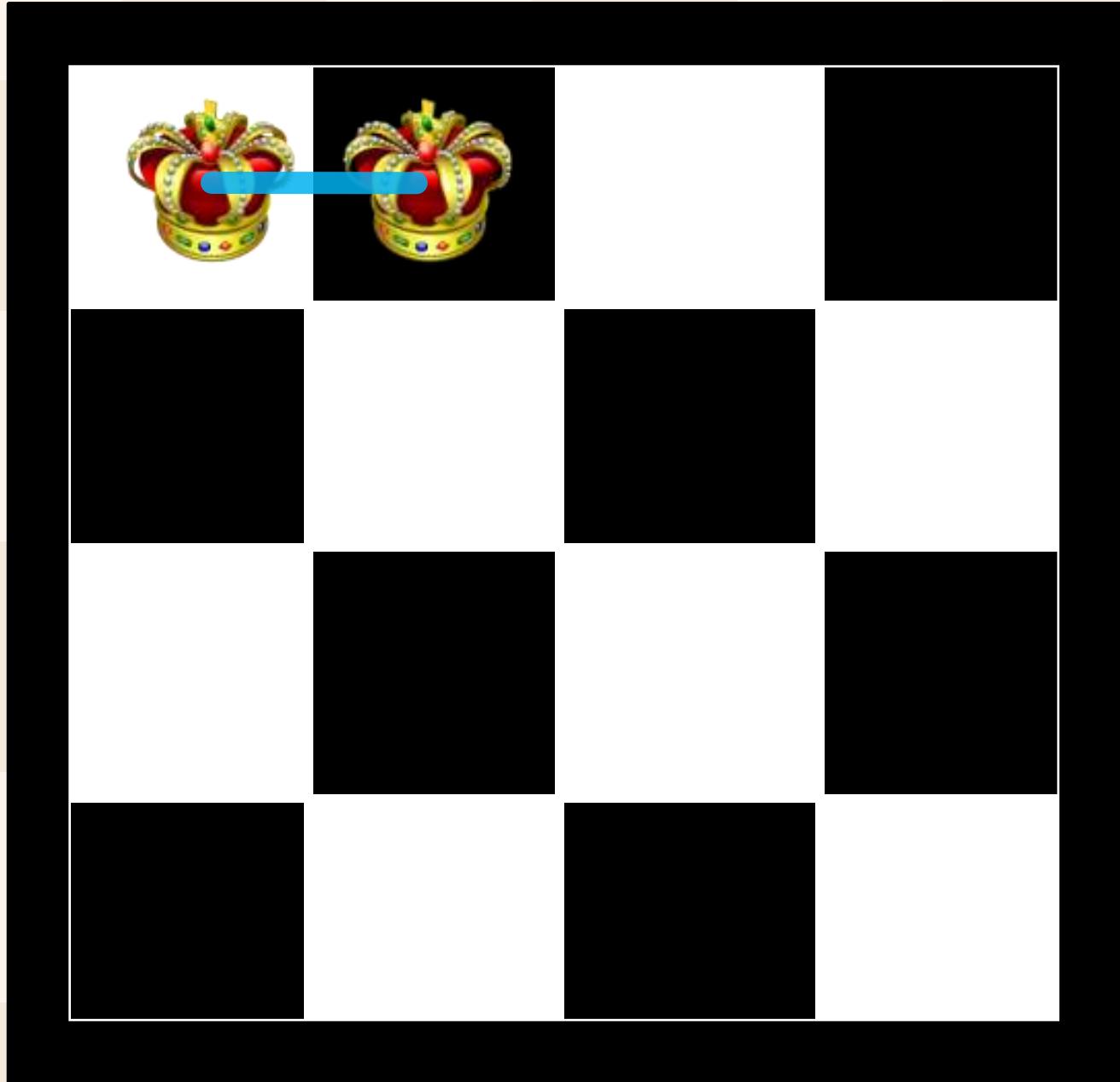
- ✓ Each recursive call attempts to place a queen in a specific column.
- ✓ For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)
- ✓ ***Current step backtracking:*** If a placement within the column does not lead to a solution, the queen is removed and moved "***down***" the column
- ✓ ***Previous step backtracking:*** When all rows in a column have been tried, the call terminates and ***backtracks to the previous call*** (in the previous column)

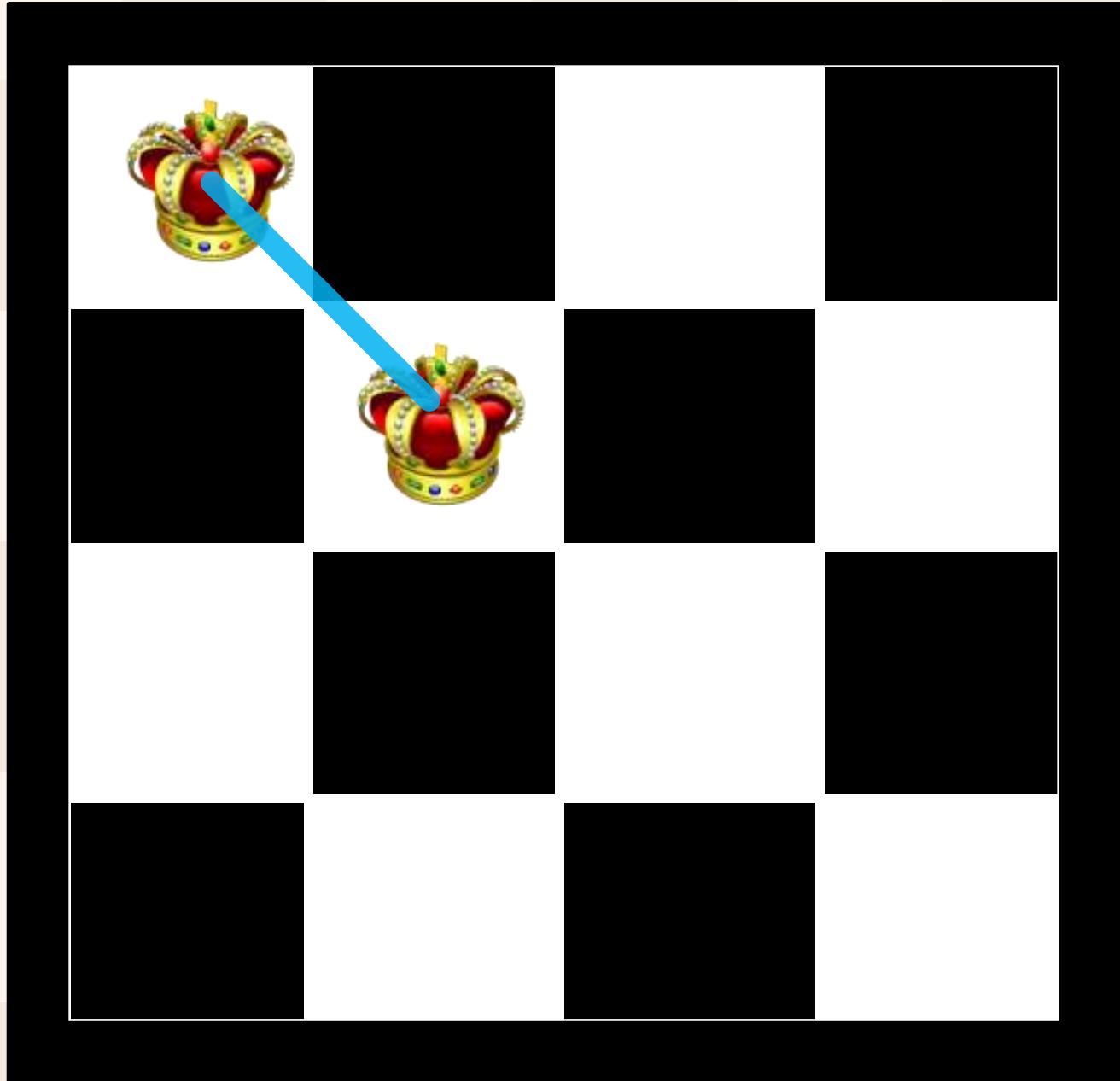
CONTINU..

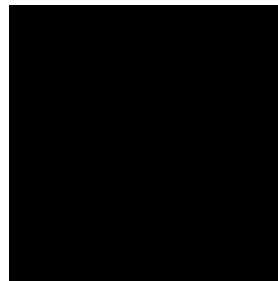
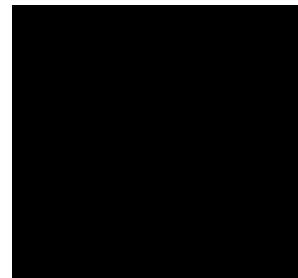
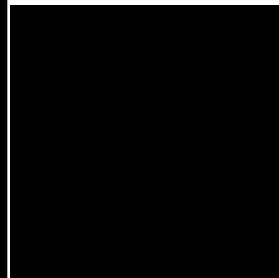
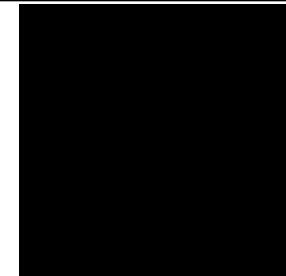
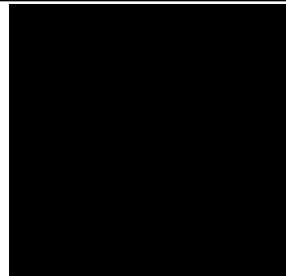
- **Pruning:** If a queen cannot be placed into column i , do not even try to place one onto column $i+1$ – rather, backtrack to column $i-1$ and move the queen that had been placed there.
- Using this approach we can reduce the number of potential solutions even more

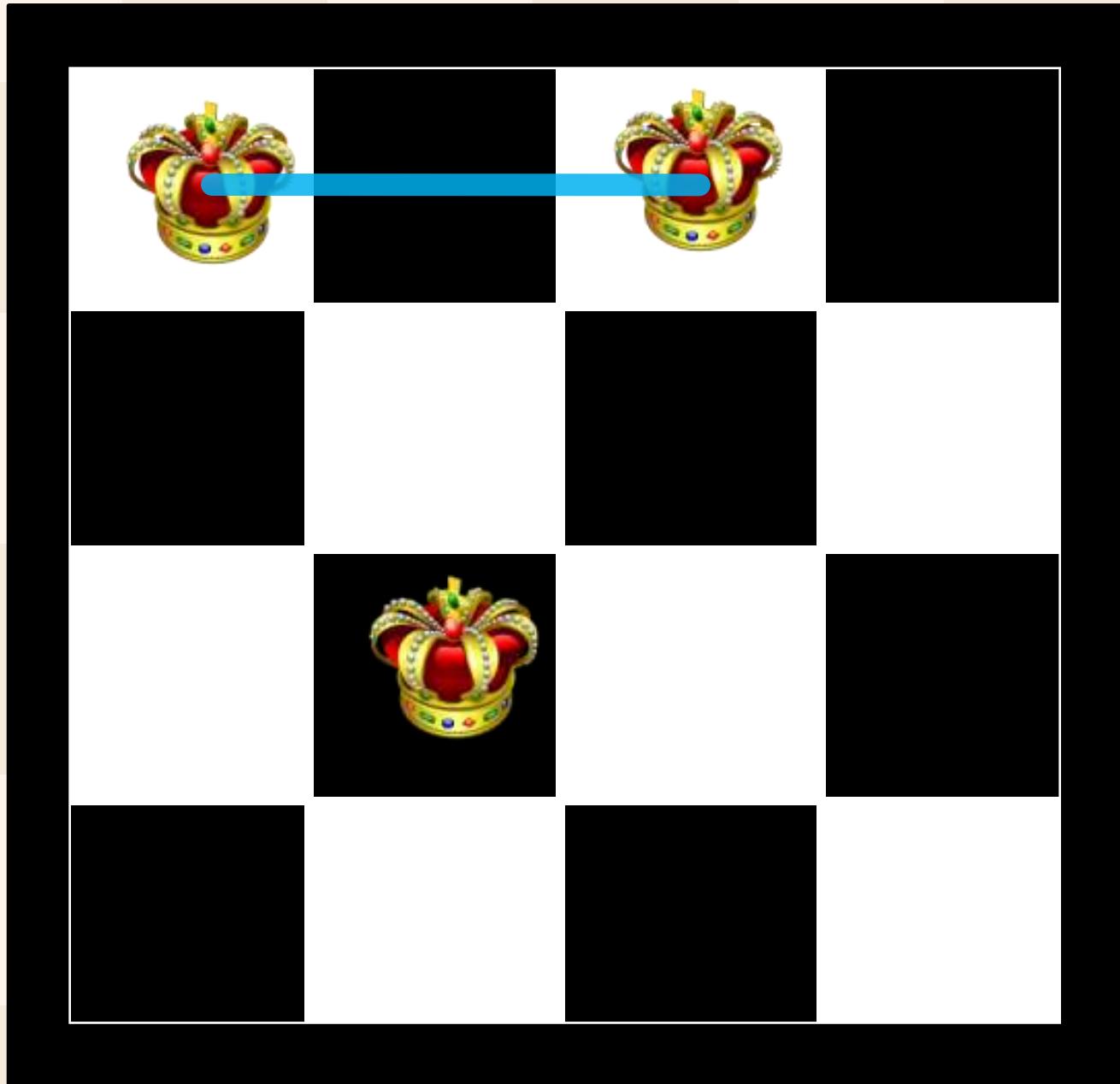
BACKTRACKING DEMO FOR 4 QUEENS

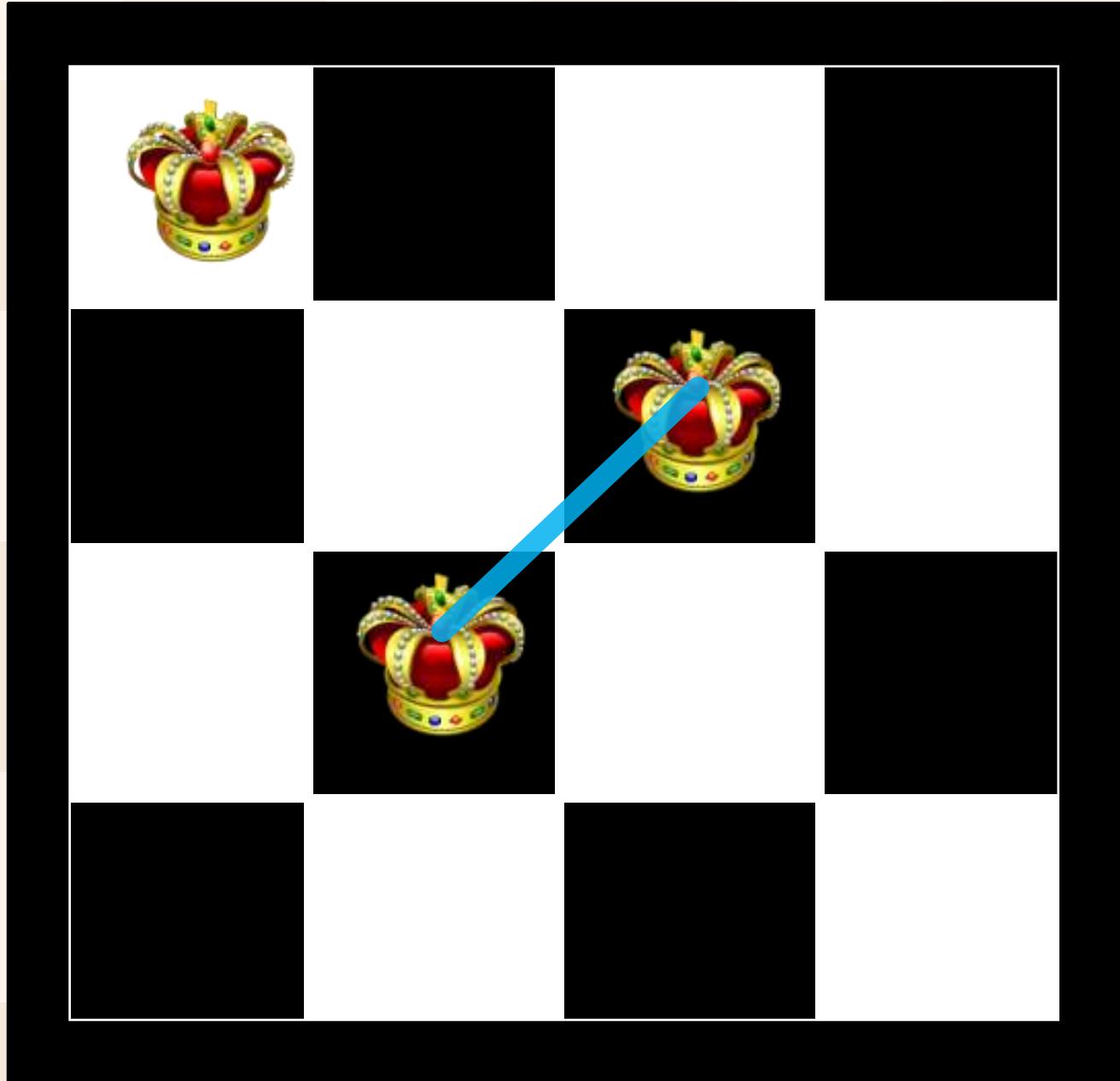


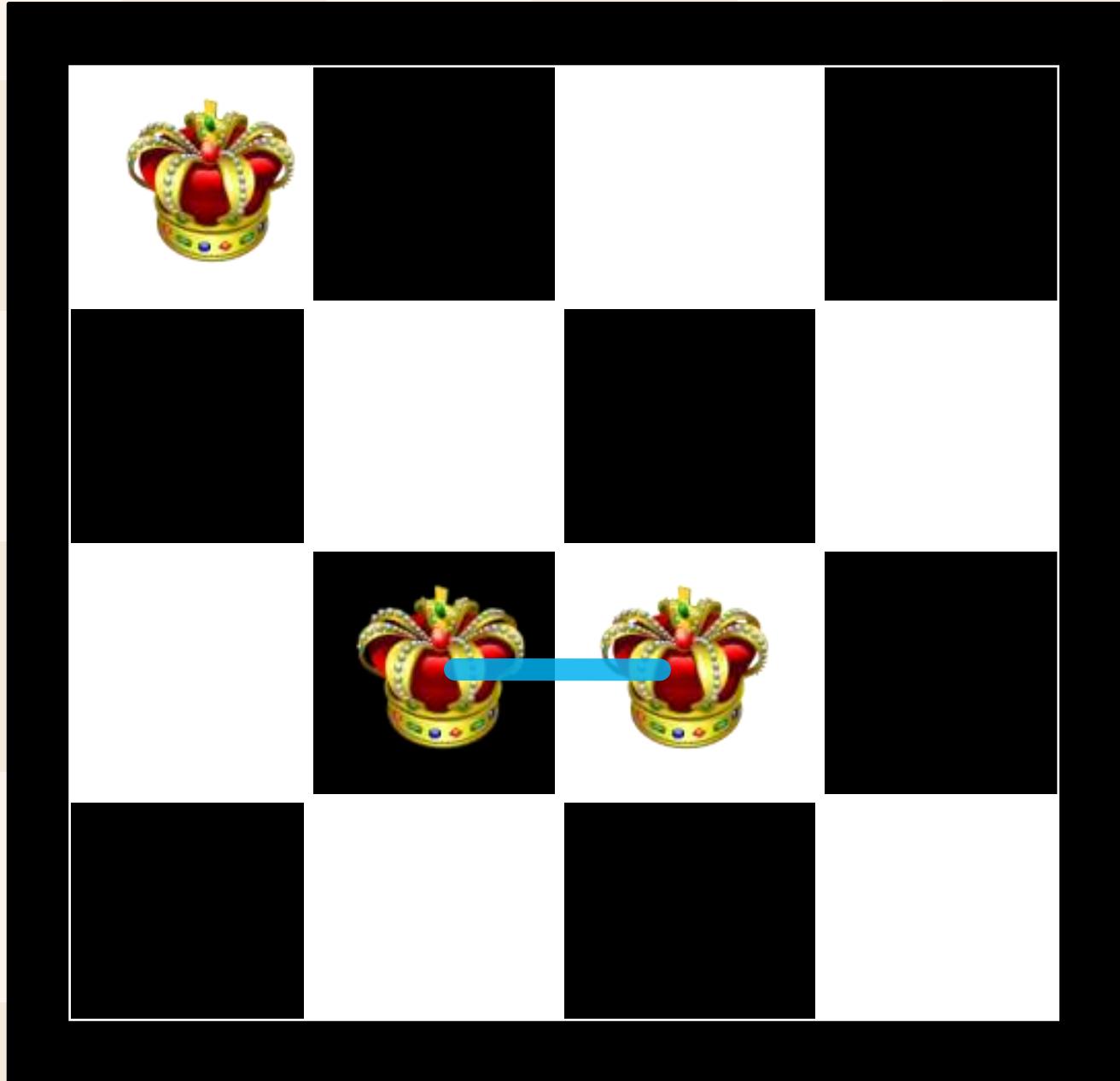


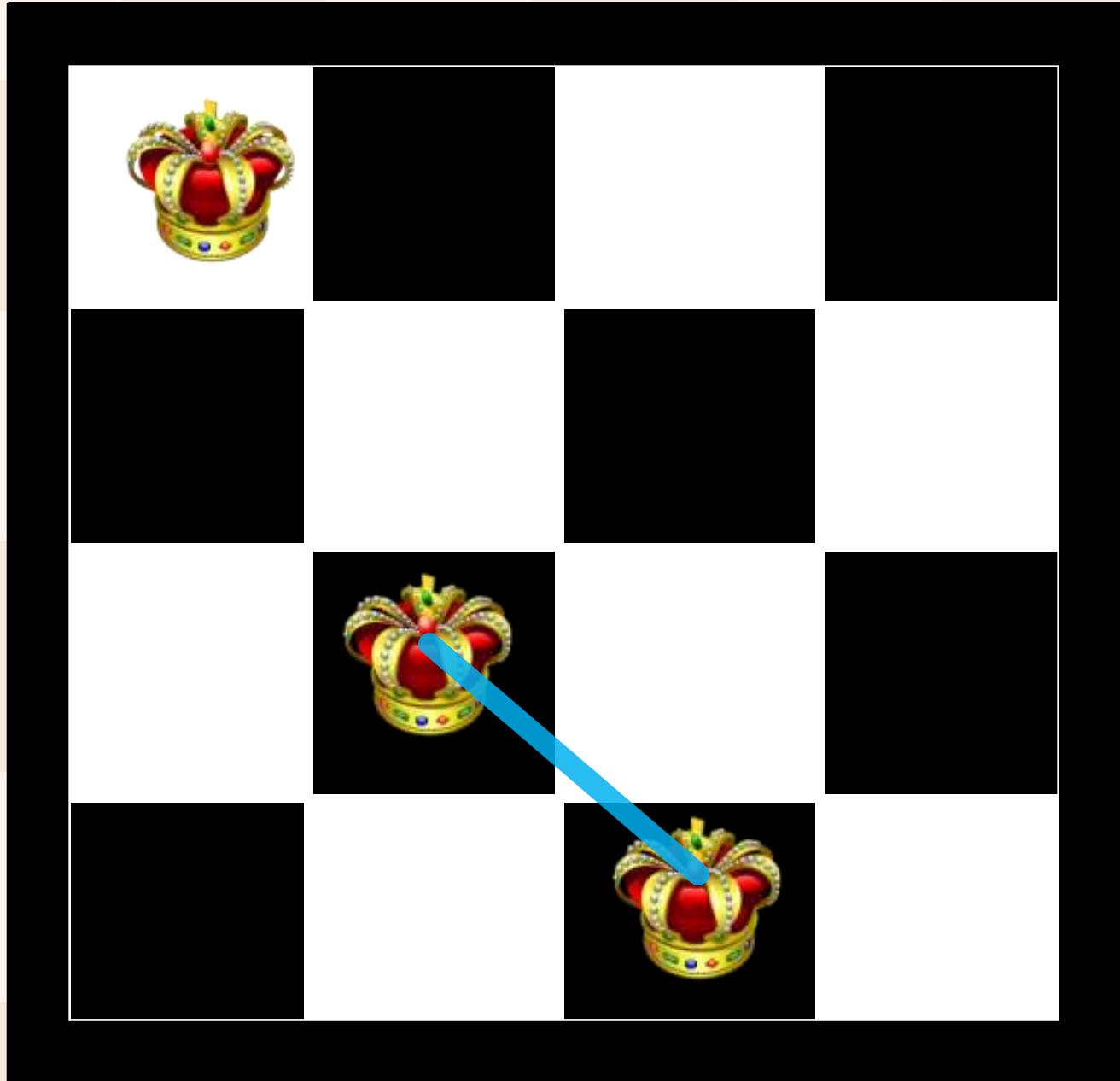


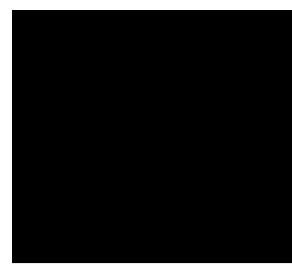
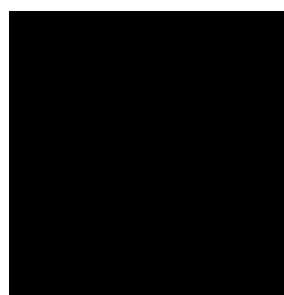
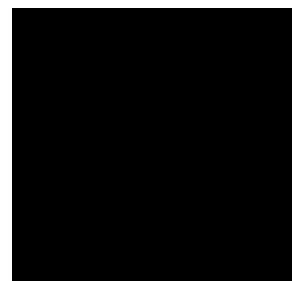
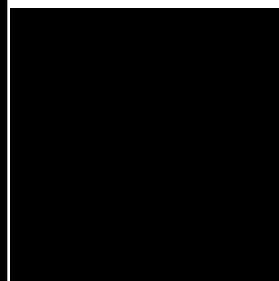
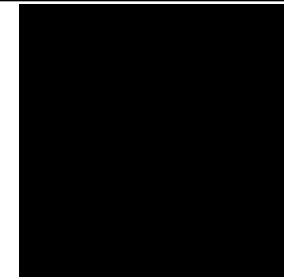
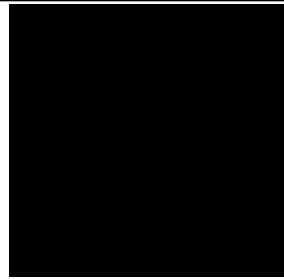


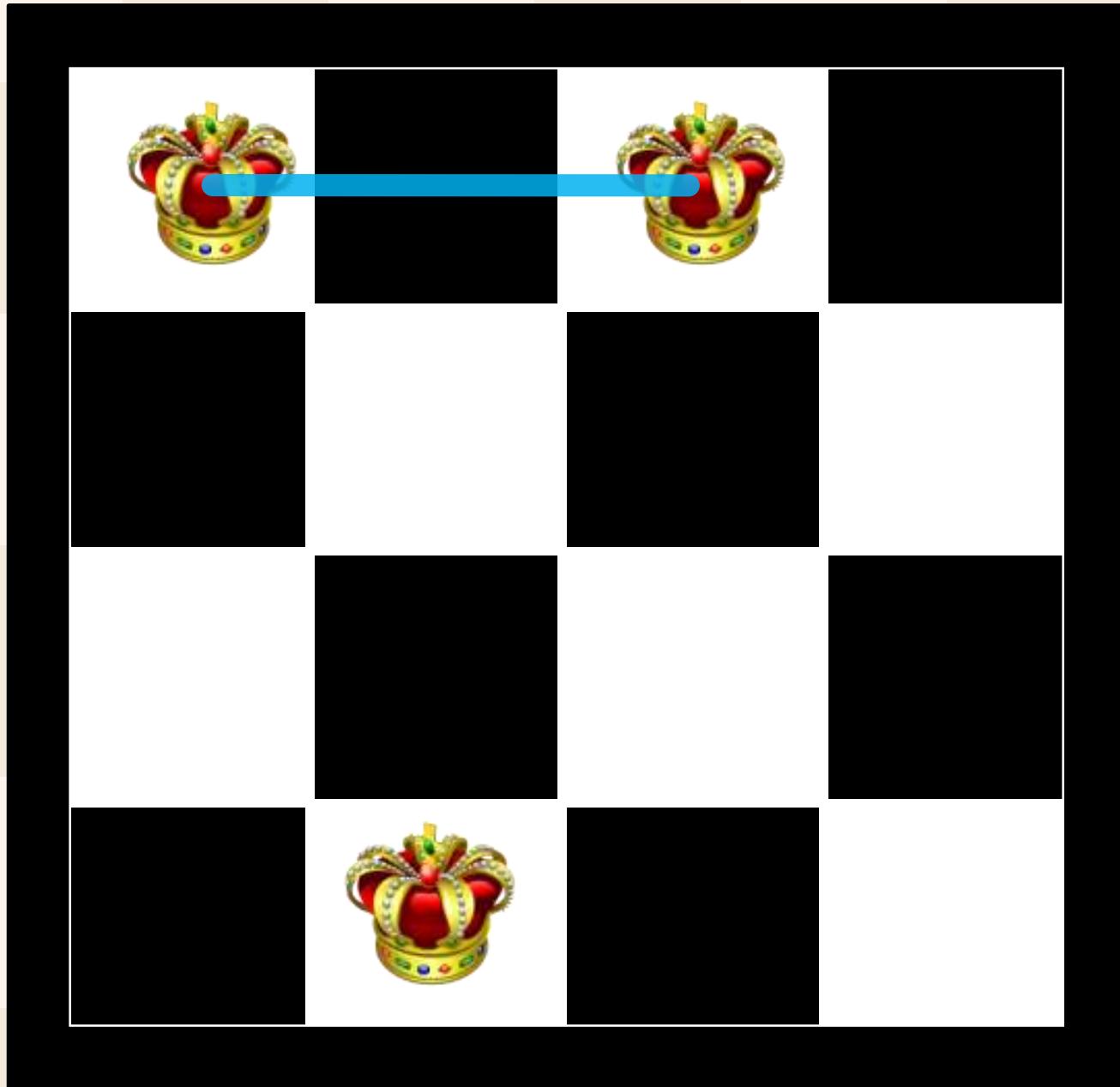




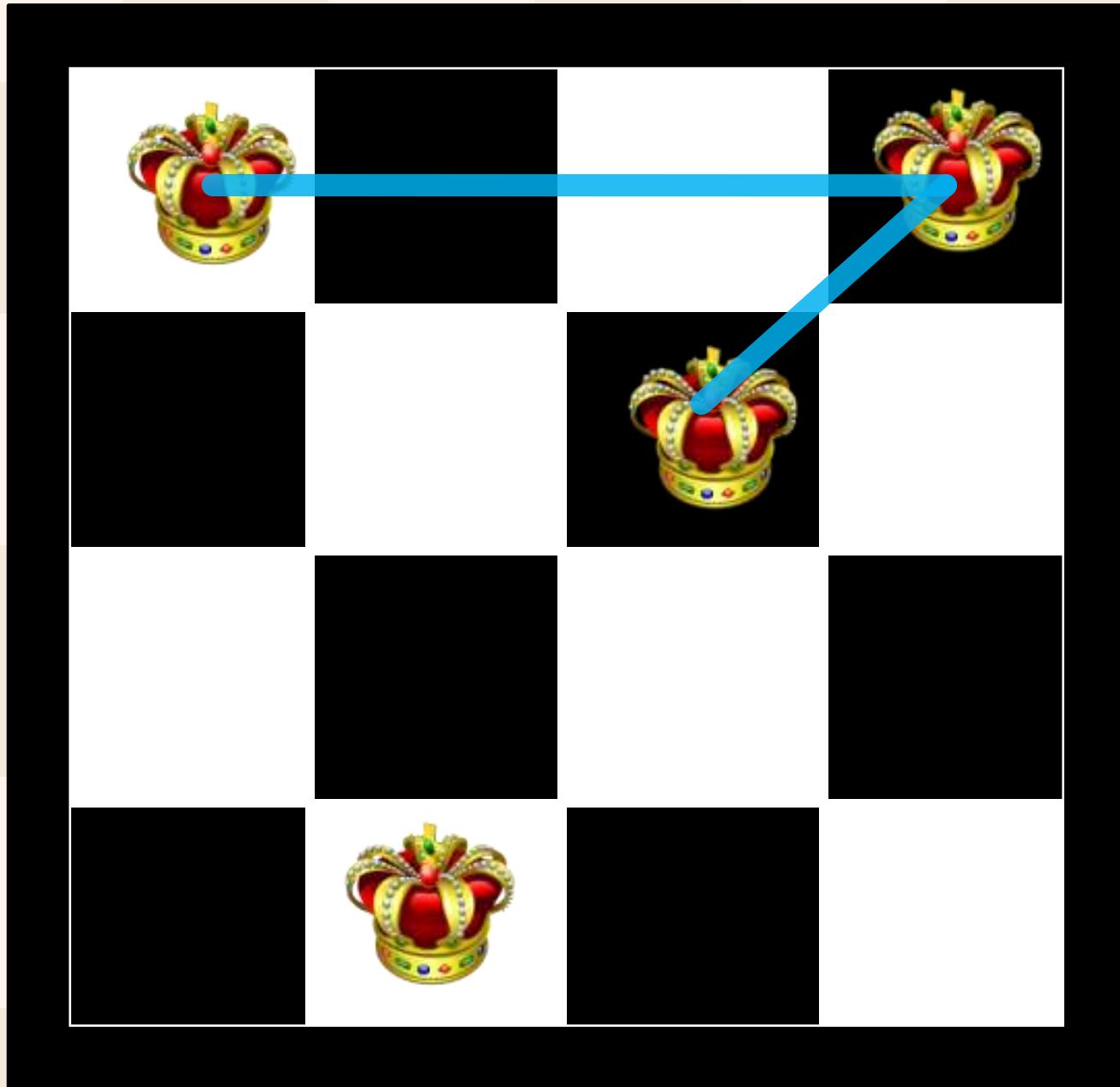


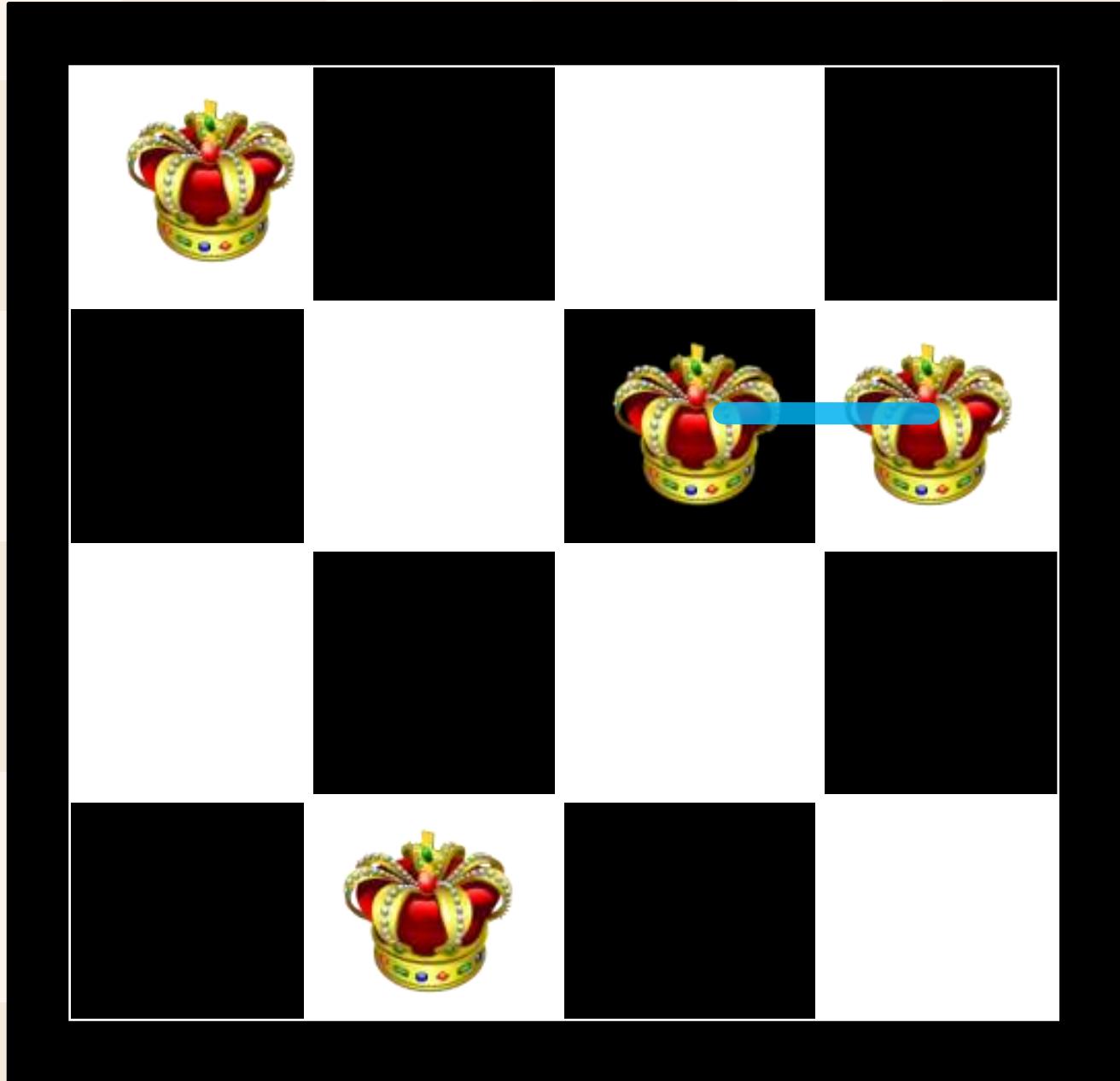


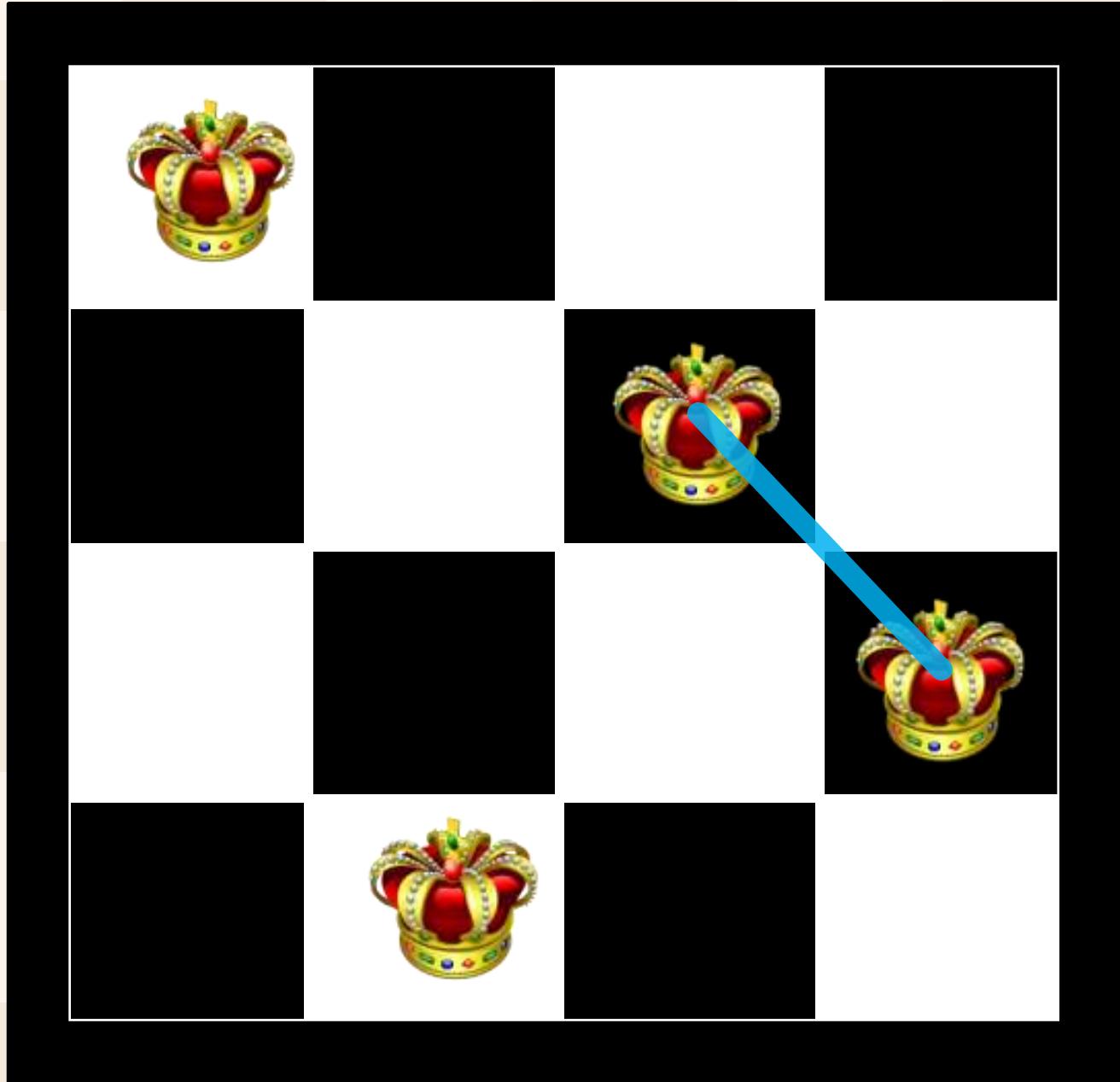


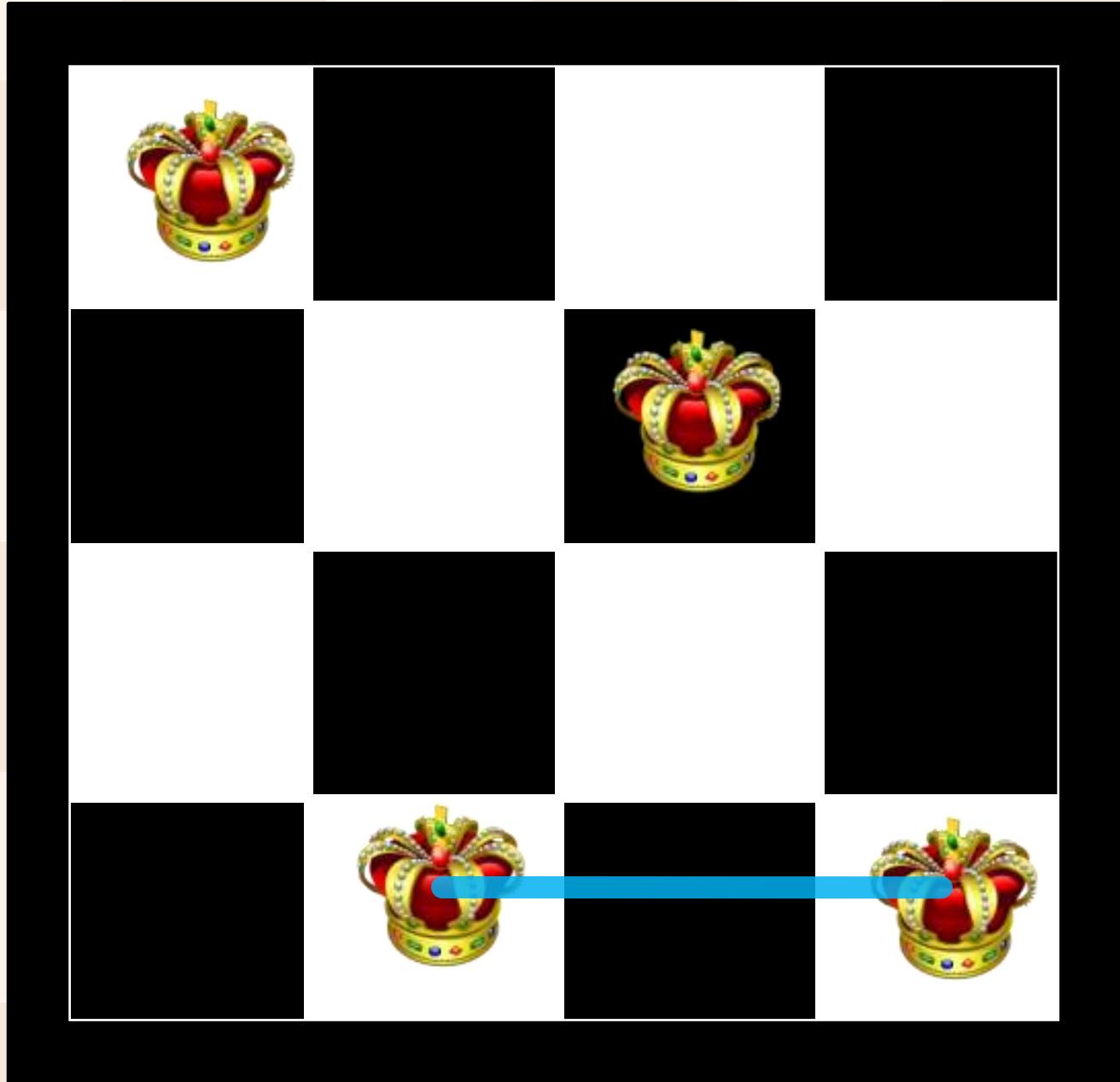


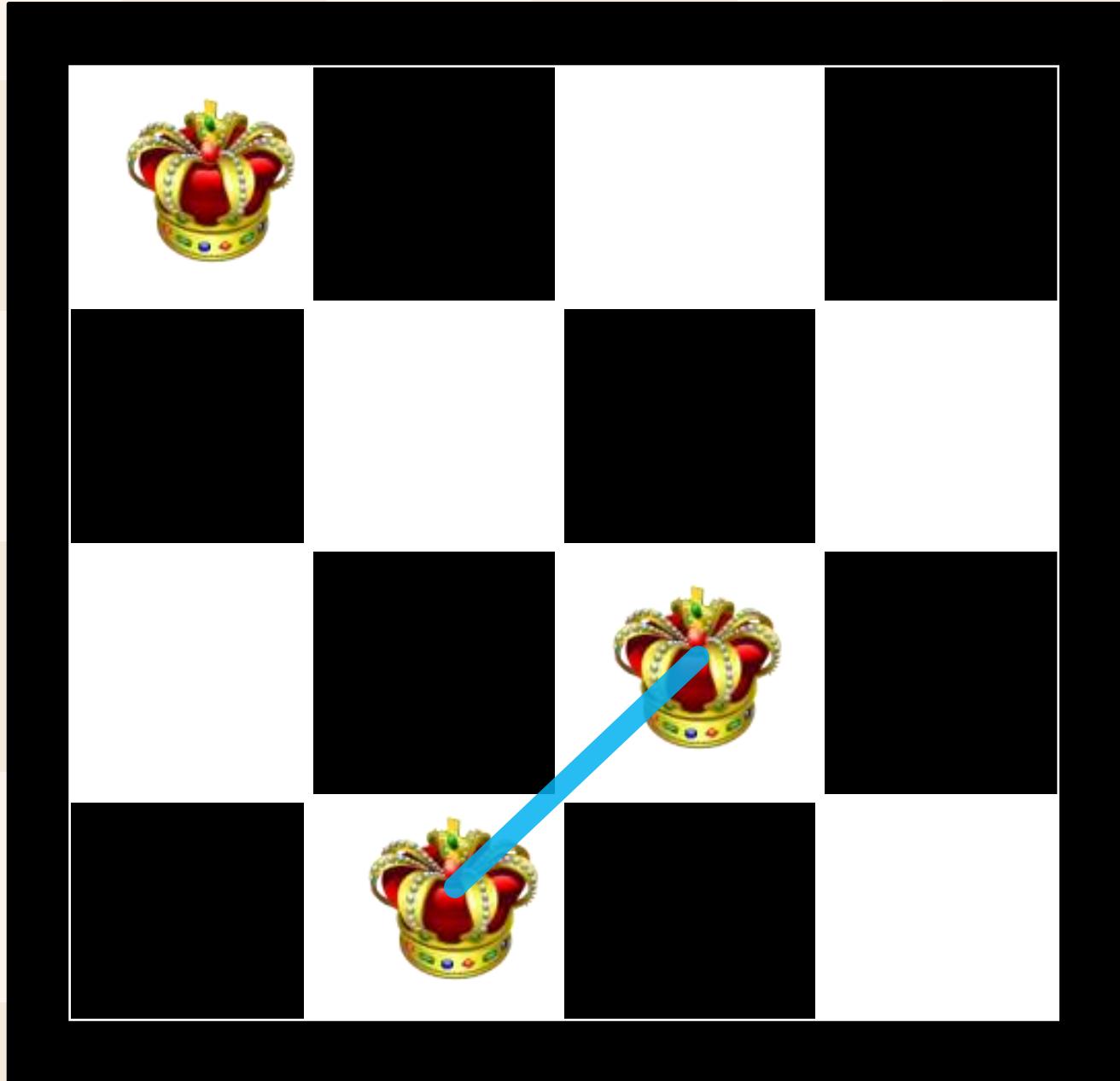


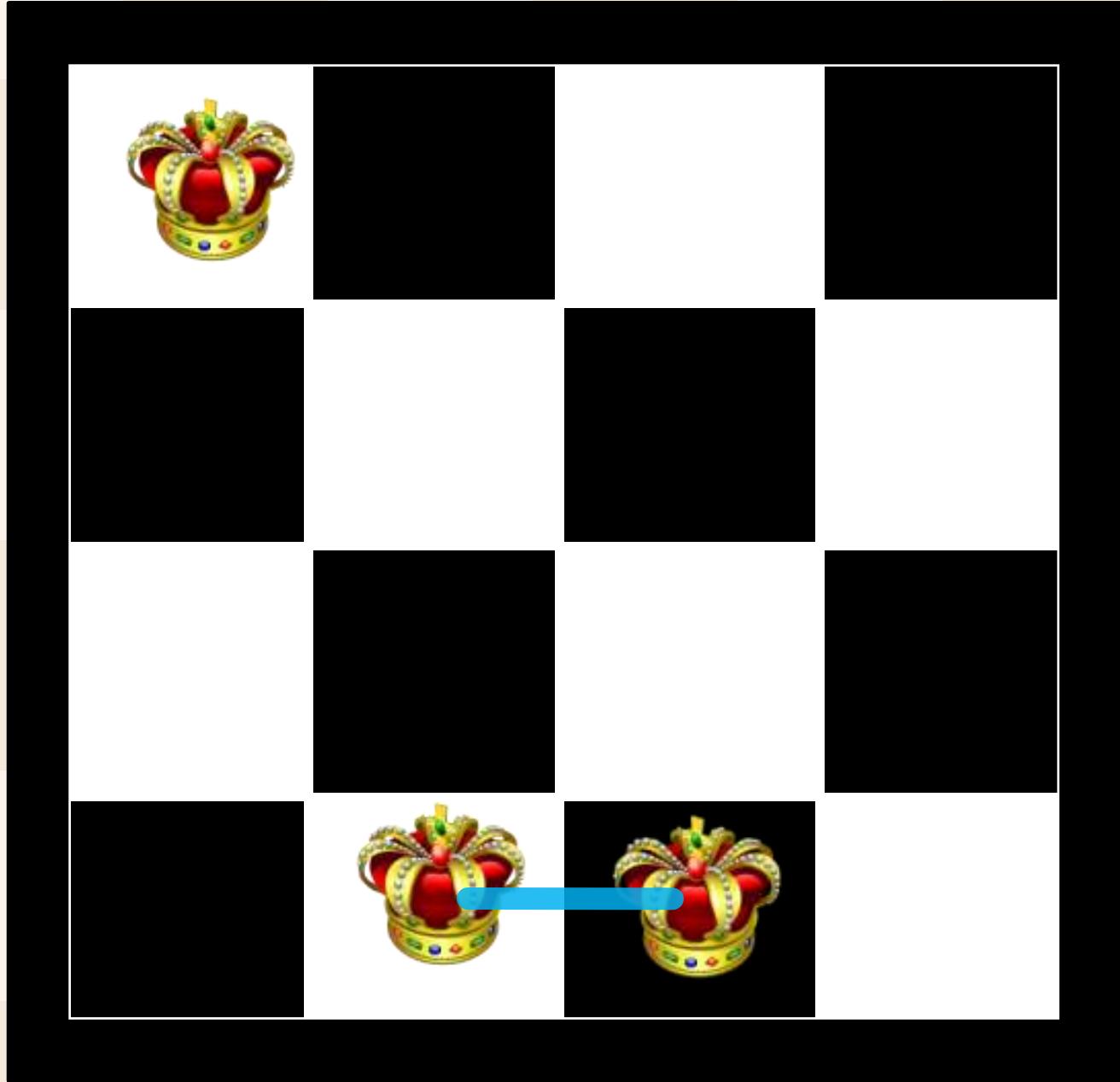


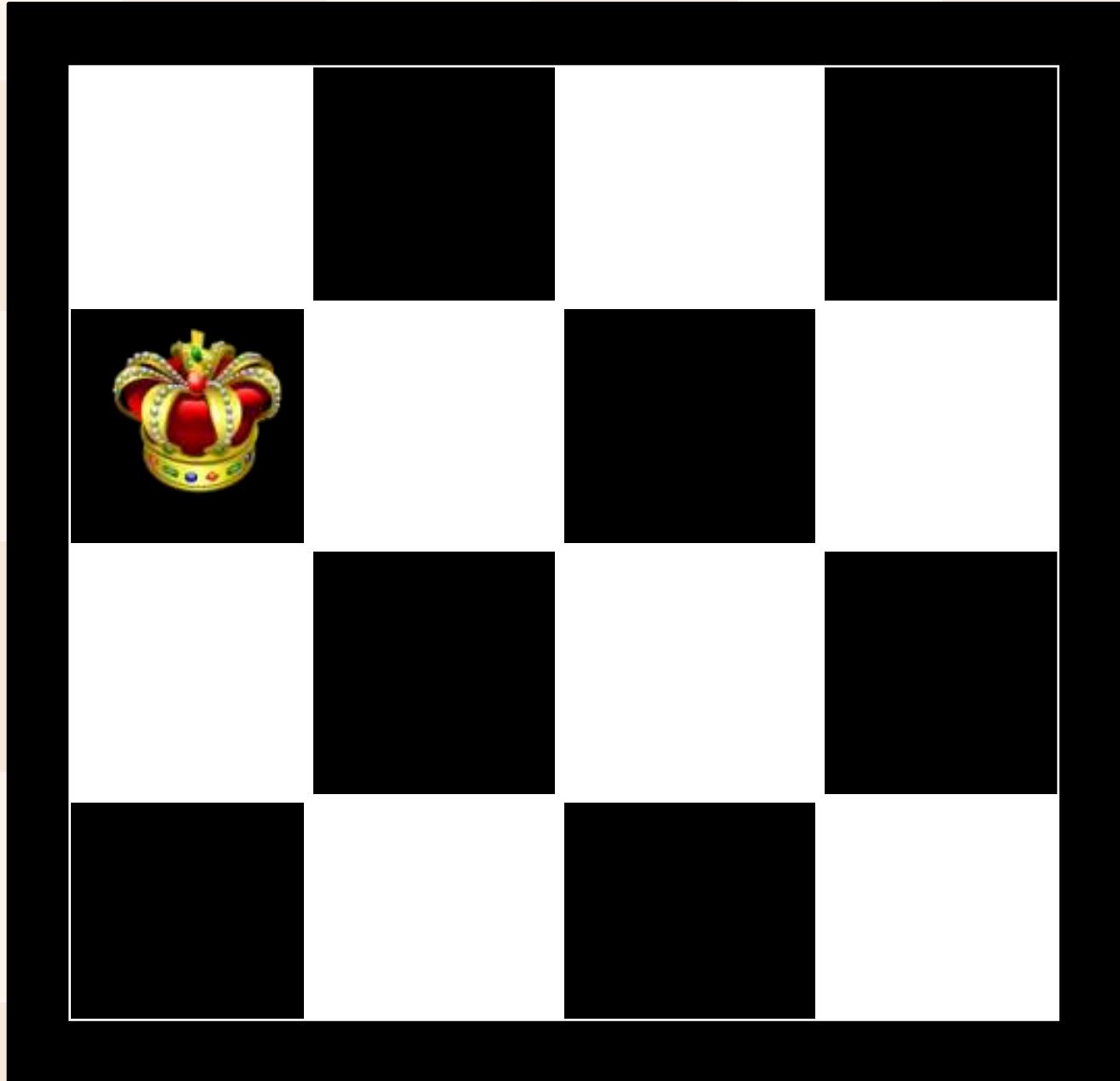


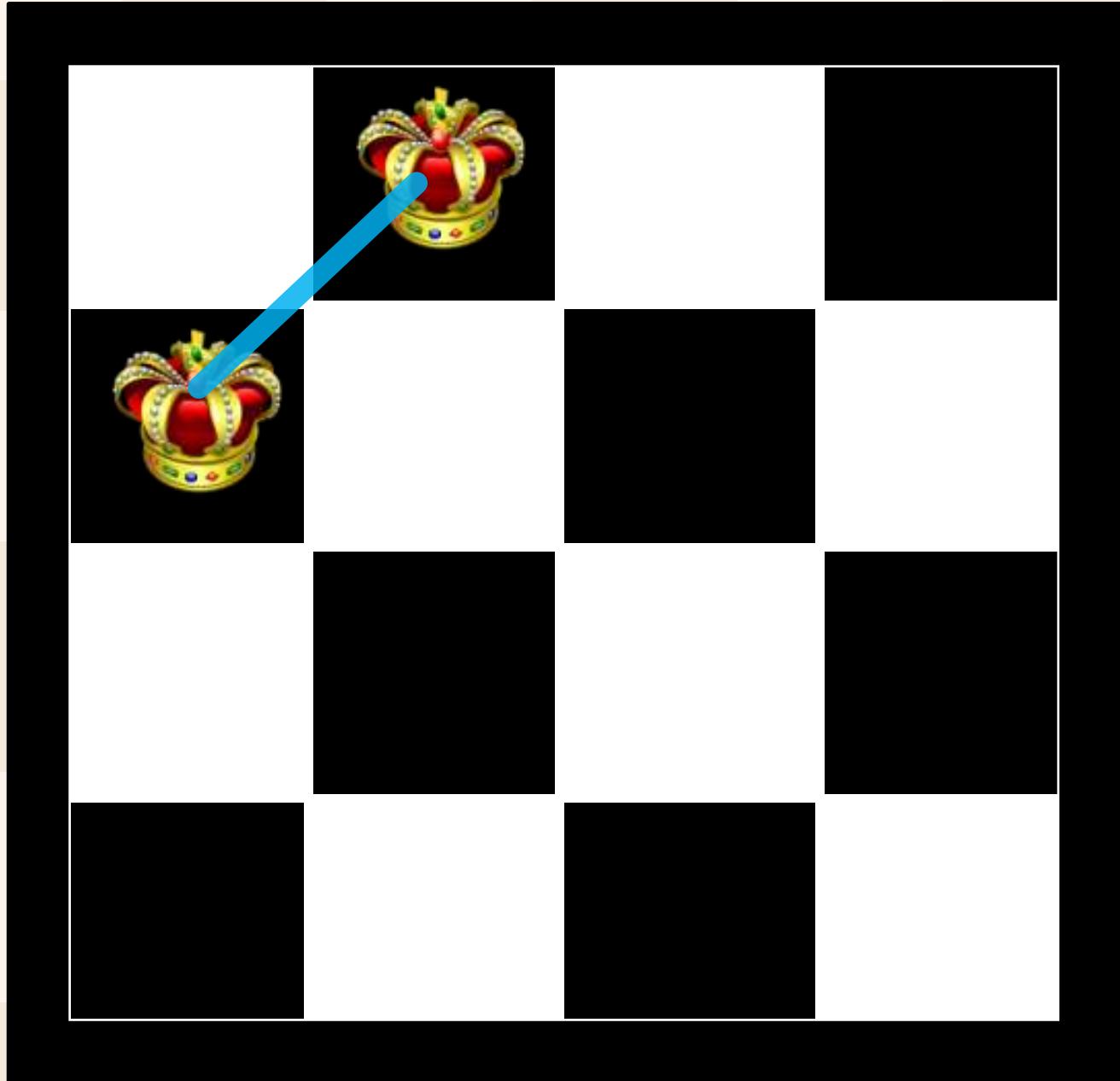


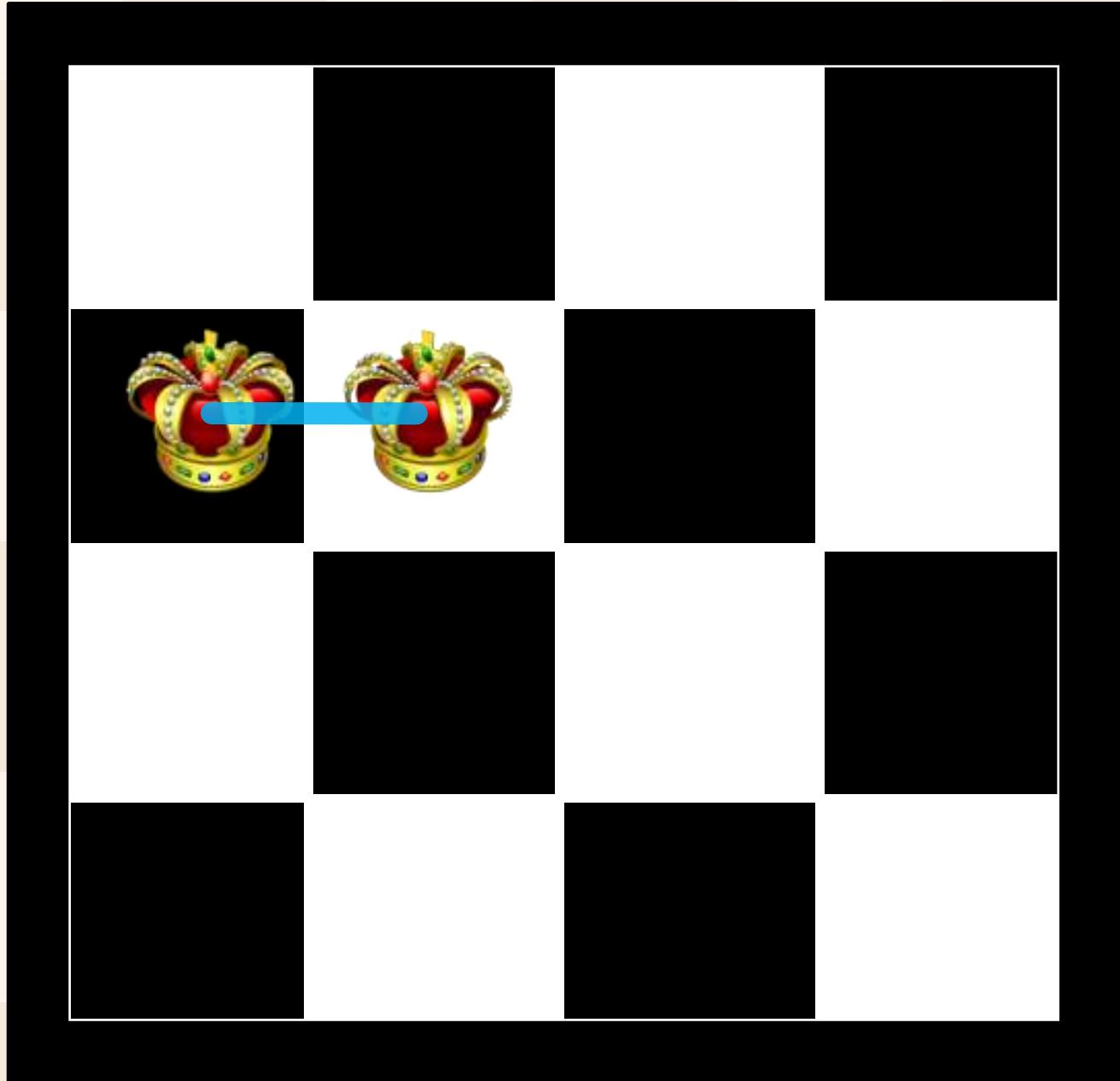


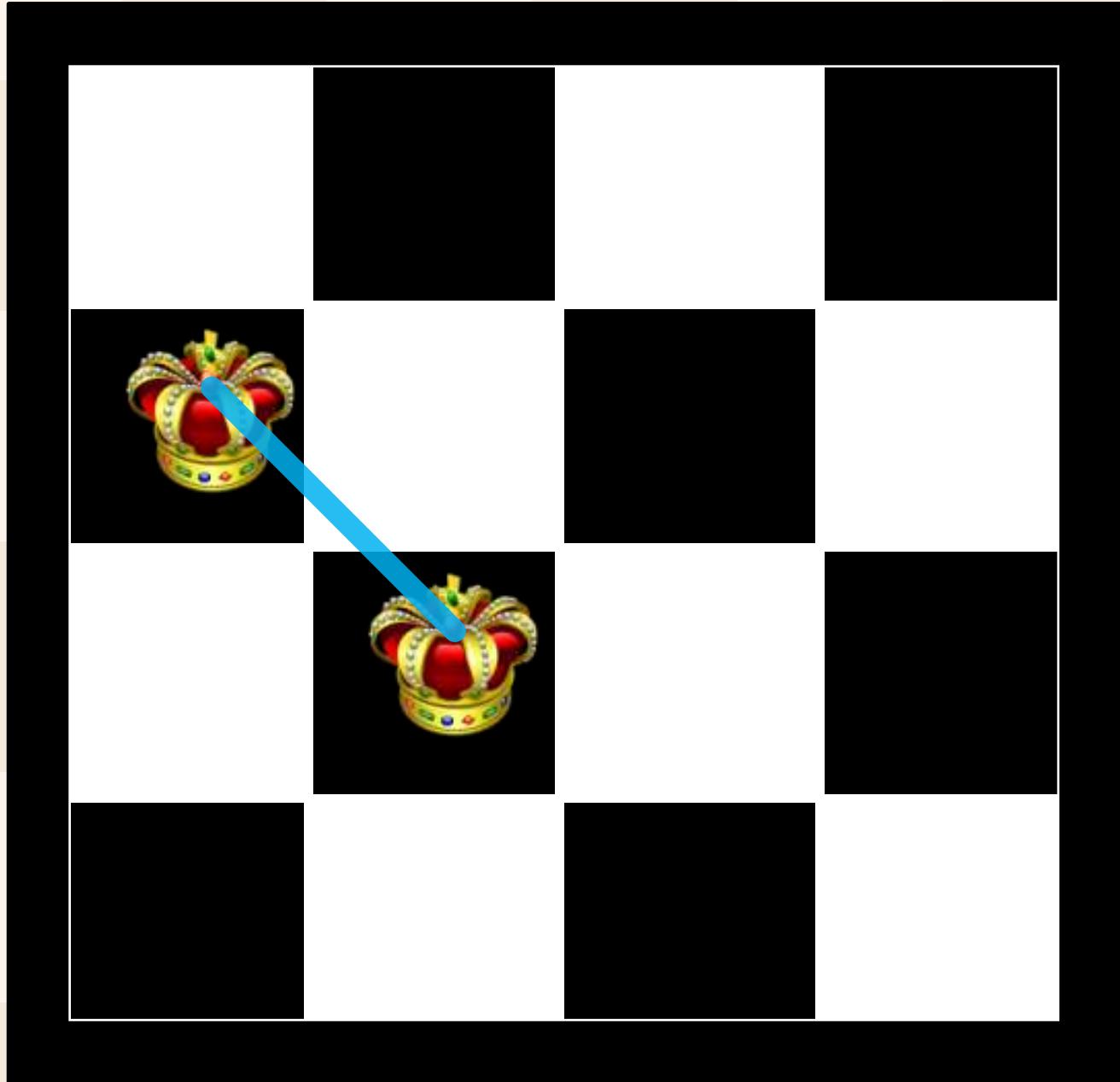


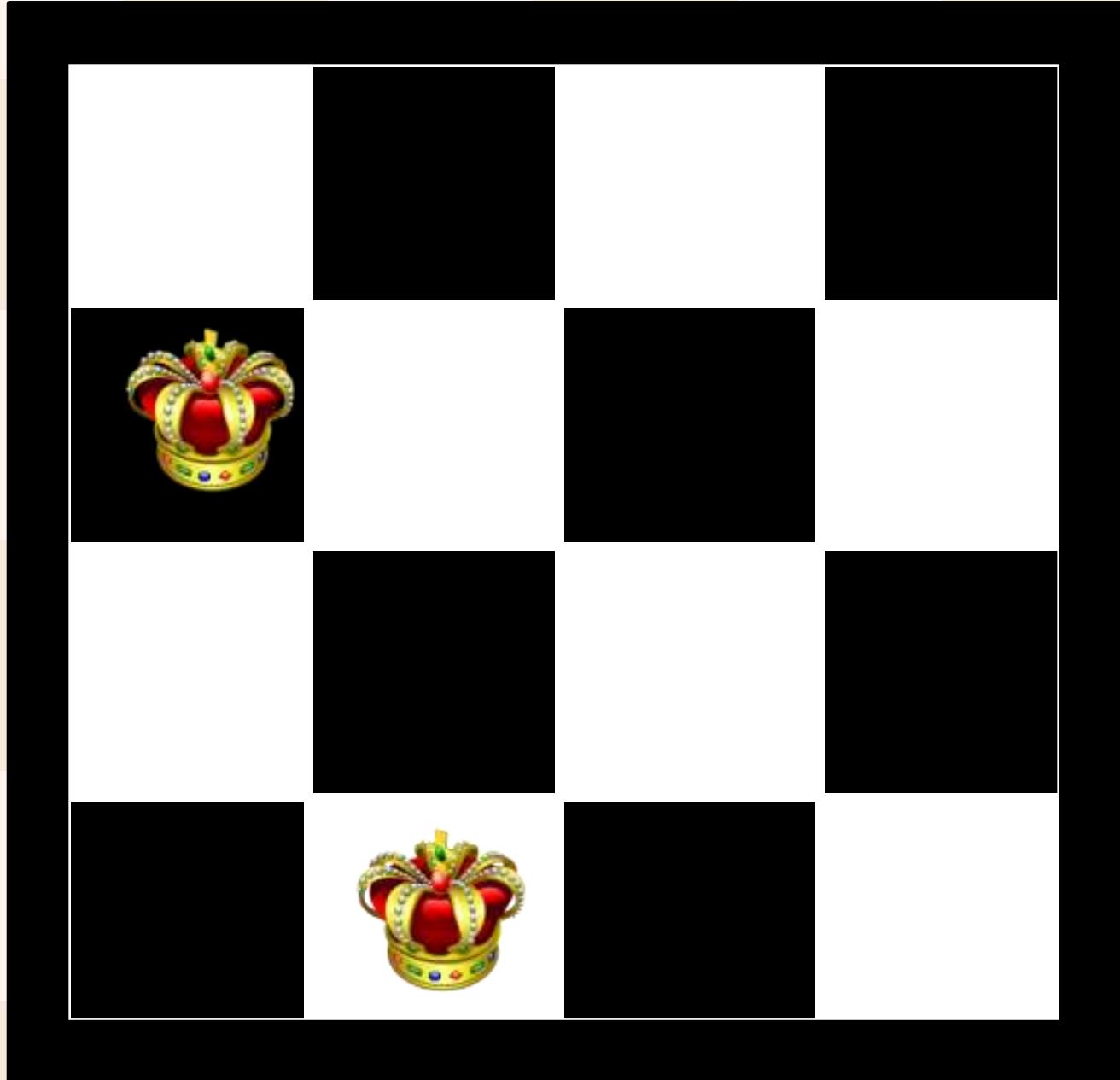


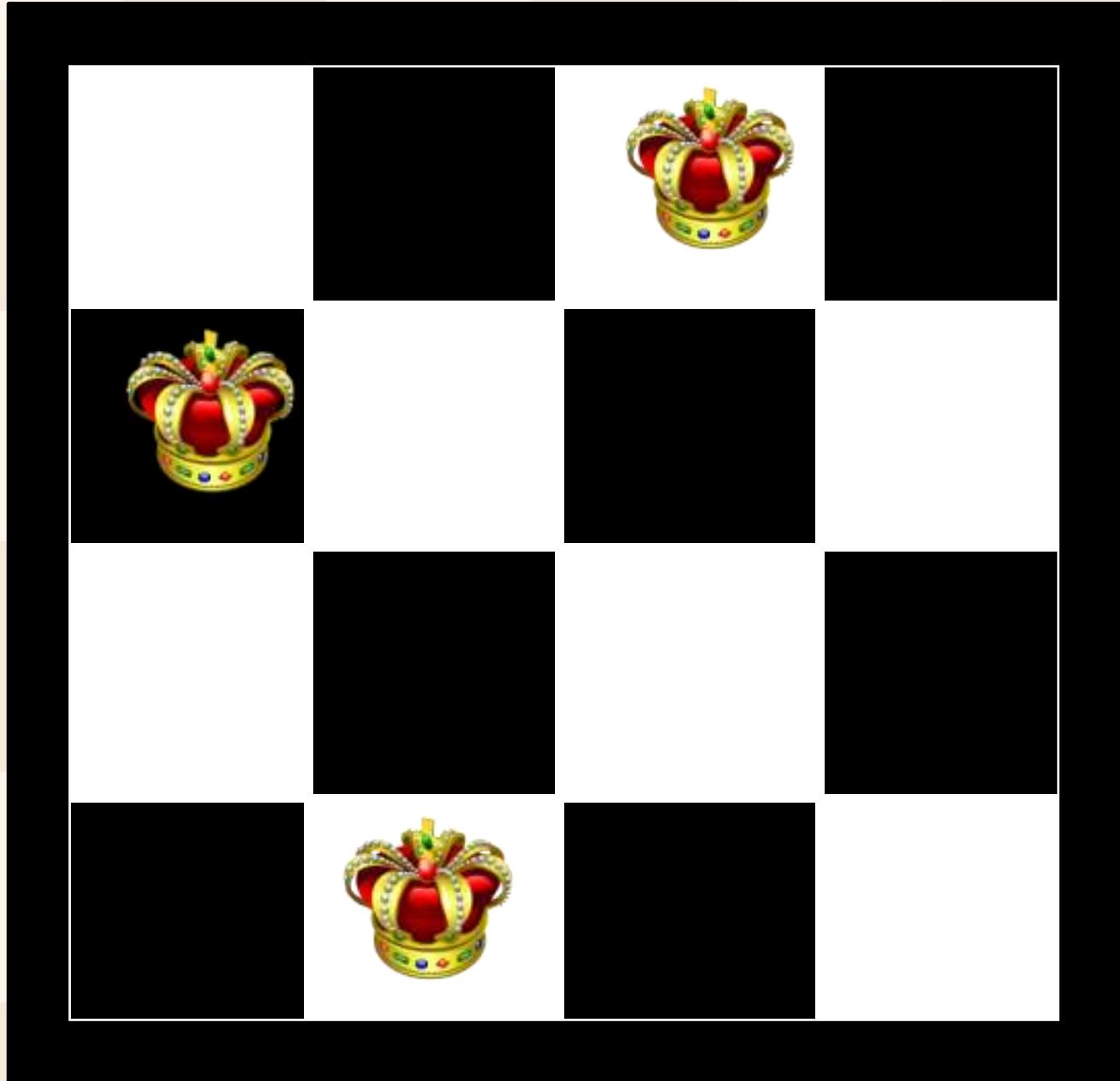


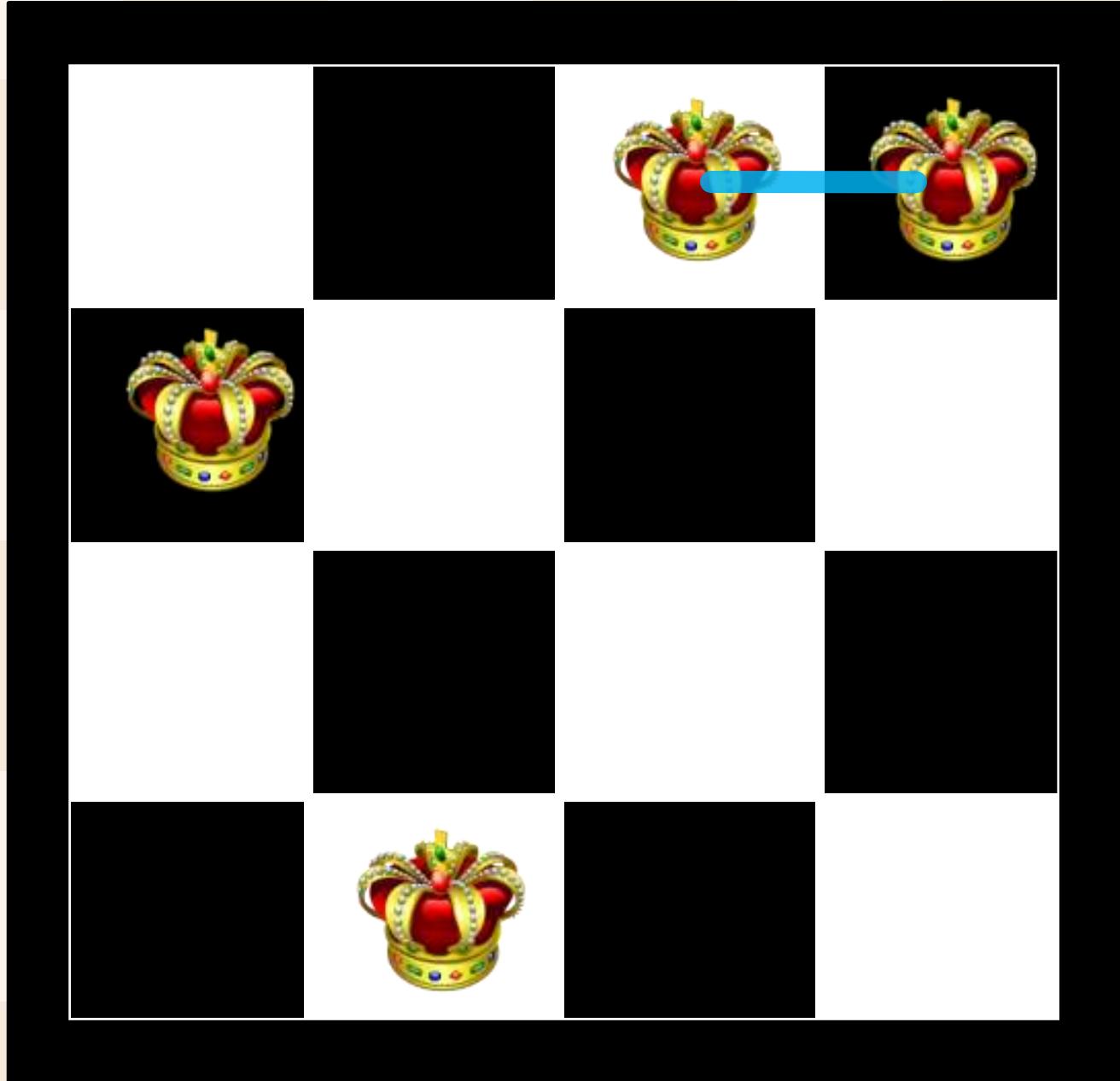


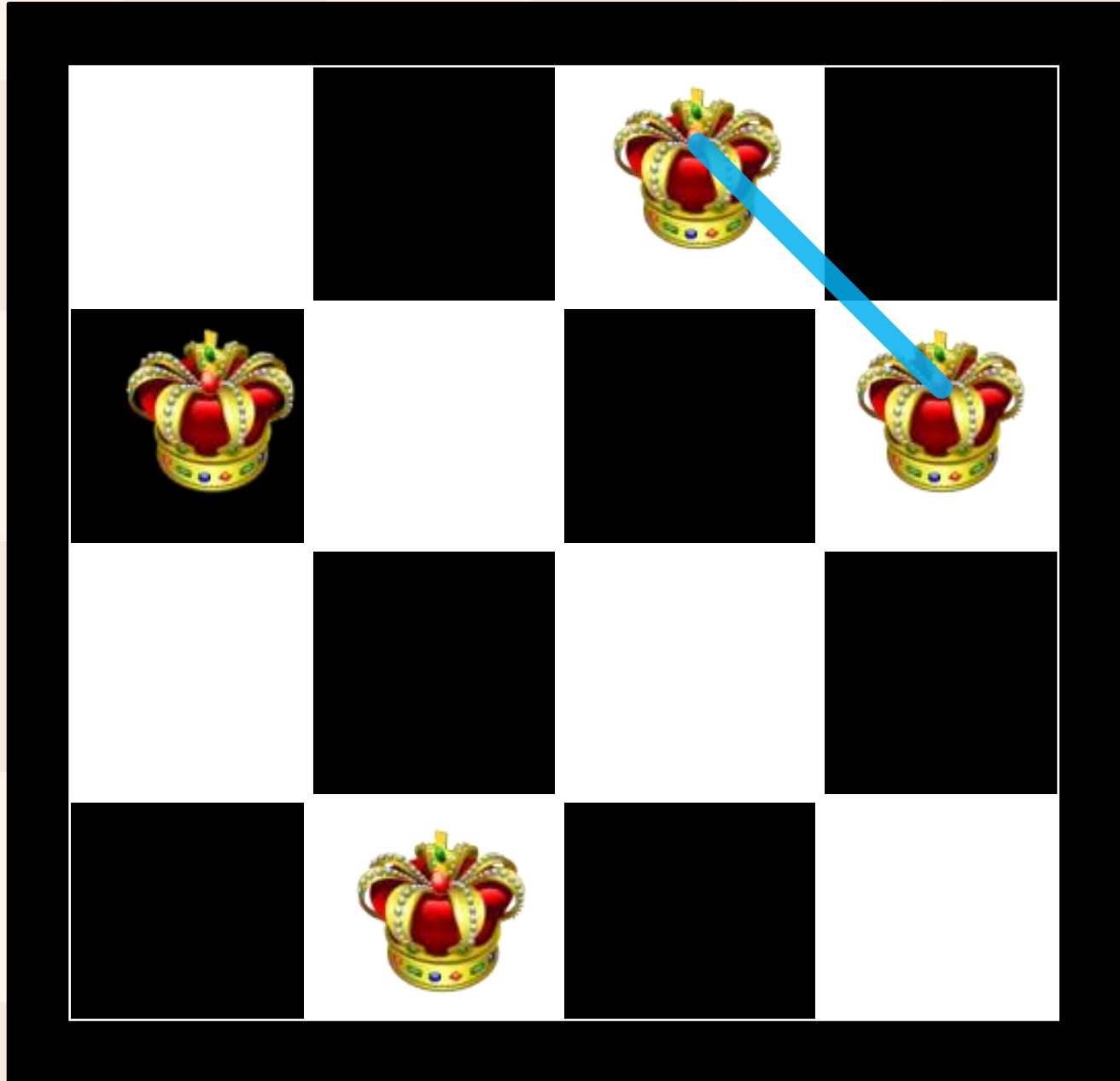


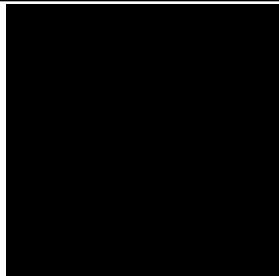






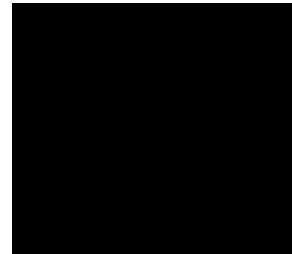
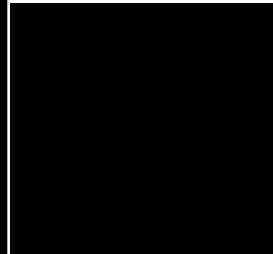
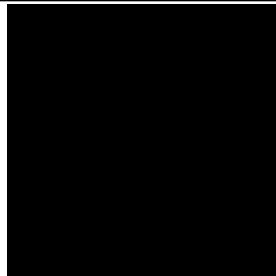






1 UNIQUE

SOLUTION



STEPS REVISITED - BACKTRACKING

1. Place the first queen in the left upper corner of the table.
2. Save the attacked positions.
3. Move to the next queen (which can only be placed to the next line).
4. Search for a valid position. If there is one go to step 8.
5. There is not a valid position for the queen. Delete it (the x coordinate is 0).
6. Move to the previous queen.
7. Go to step 4.
8. Place it to the first valid position.
9. Save the attacked positions.
10. If the queen processed is the last stop otherwise go to step 3.

EIGHT QUEEN PROBLEM: ALGORITHM

putQueen(row)

{

for every position col on the same row

 if position col is available

 place the next queen in position col

 if (row<8)

 putQueen(row+1);

 else success;

 remove the queen from position col

}

THE PUTQUEEN RECURSIVE METHOD

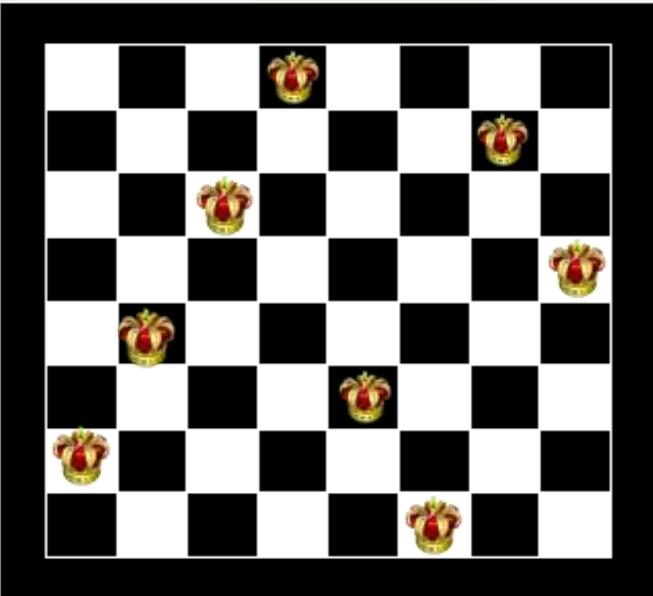
```
void putQueen(int row)
{
    for (int col=0;col<squares;col++)
        if (column[col]==available &&
            leftDiagonal[row+col]==available &&
            rightDiagonal[row-col]== available)
        {
            positionInRow[row]=col;
            column[col]=!available;
            leftDiagonal[row+col]=!available;
```

```
rightDiagonal[row-col]=!available;
if (row< squares-1)
    putQueen(row+1);
else
    print(" solution found");
column[col]=available;
leftDiagonal[row+col]=available;
rightDiagonal[row-col]= available;
}
}
```

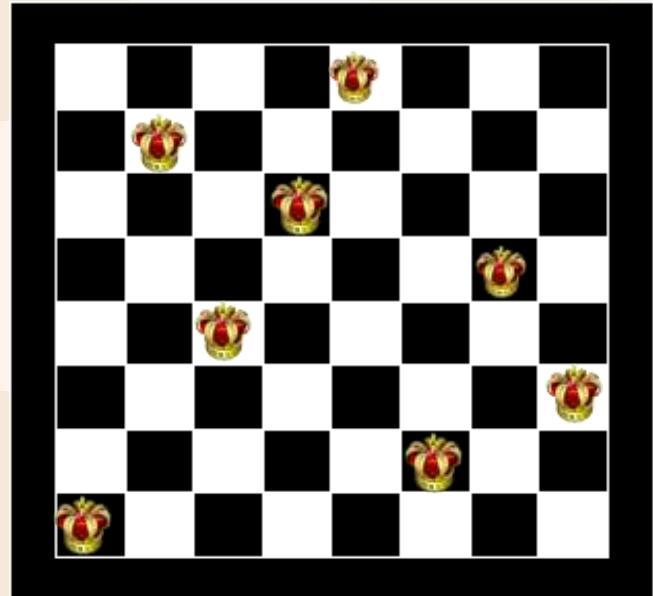
SOLUTIONS

- The eight queens puzzle has 92 **distinct** solutions.
- If solutions that differ only by symmetry operations(rotations and reflections) of the board are counted as one the puzzle has 12 **unique** (or **fundamental**) solutions

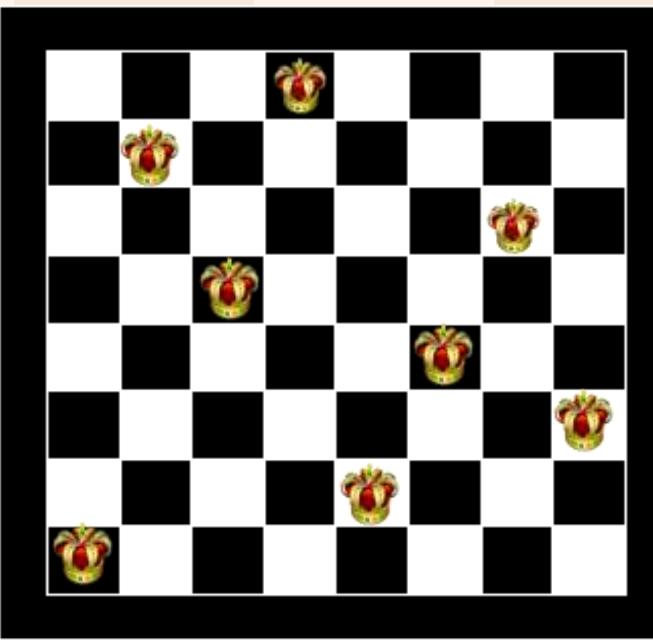
UNIQUE SOLUTION 1



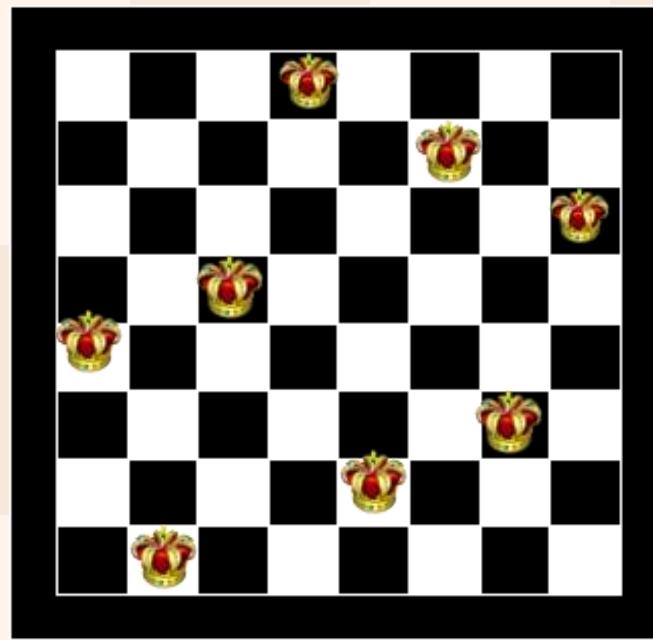
UNIQUE SOLUTION 2



UNIQUE SOLUTION 3



UNIQUE SOLUTION 4



COUNTING SOLUTIONS

- ✓ The following table gives the number of solutions for placing n queens on an $n \times n$ board, both unique and distinct for $n=1-26$.
- ✓ Note that the six queens puzzle has fewer solutions than the five queens puzzle.
- ✓ There is currently no known formula for the exact number of solutions.

Order ("N")	Total Solutions	Unique Solutions	Exec time
1	1	1	< 0 seconds
2	0	0	< 0 seconds
3	0	0	< 0 seconds
4	2	1	< 0 seconds
5	10	2	< 0 seconds
6	4	1	< 0 seconds
7	40	6	< 0 seconds
8	92	12	< 0 seconds
9	352	46	< 0 seconds
10	724	92	< 0 seconds
11	2,680	341	< 0 seconds
12	14,200	1,787	< 0 seconds
13	73,712	9,233	< 0 seconds
14	365,596	45,752	0.2s

15	2,279,184	285,053	1.9 s
16	14,772,512	1,846,955	11.2 s
17	95,815,104	11,977,939	77.2 s
18	666,090,624	83,263,591	9.6 m
19	4,968,057,848	621,012,754	75.0 m
20	39,029,188,884	4,878,666,808	10.2 h
21	314,666,222,712	39,333,324,973	87.2 h
22	2,691,008,701,644	336,376,244,042	31.9
23	24,233,937,684,440	3,029,242,658,210	296 d
24	227,514,171,973,736	28,439,272,956,934	?
25	2,207,893,435,808,352	275,986,683,743,434	?
26	22,317,699,616,364,044	2,789,712,466,510,289	?

(s = seconds m = minutes h = hours d = days)

JEFF SOMER'S ALGORITHM

- ✓ His algorithm for the N-Queen problem is considered as the fastest algorithm. He uses the concept of back tracking to solve this
- ✓ Previously the World's fastest algorithm for the N-Queen problem was given by **Sylvain Pion and Joel-Yann Fourre**.
- ✓ His algorithm finds solutions up to 23 queens and uses bit field manipulation in **BACKTRACKING**.
- ✓ According to his program the maximum time taken to find all the solutions for a 18 queens problem is 00:19:26 where as in the normal back tracking algorithm it was 00:75:00.

USING NESTED LOOPS FOR SOLUTION

For a 4x4 board, we could find the solutions like this:

```
for(i0 = 0; i0 < 4; ++i0)
{
    if(isSafe(board, 0, i0))
    {
        board[0][i0] = true;
        for(i1 = 0; i1 < 4; ++i1)
        {
            if(isSafe(board, 1, i1))
            {
                board[1][i1] = true;
                for(i2 = 0; i2 < 4; ++i2)
                {
                    if(isSafe(board, 2, i2))
                    {
                        board[2][i2] = true;
                        for(i3 = 0; i3 < 4; ++i3)
                        {
                            if(isSafe(board, 3, i3))
                            {
                                board[3][i3] = true;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
{  
    printBoard(board, 4);  
}  
    board[3][i3] = false; }  
}  
    board[2][i2] = false; }  
}  
    board[1][i1] = false; }  
}  
    board[0][io] = false; }
```

WHY NOT NESTED LOOP

- ✓ The nested loops are not so preferred because . It Does not scale to different sized boards
- ✓ You must duplicate identical code (place and remove). and error in one spot is hard to find
- ✓ The problem with this is that it's not very programmer-friendly. We can't vary at runtime the size of the board we're searching

- ✓ The major advantage of the backtracking algorithm is the ability to find and count all the possible solutions rather than just one while offering decent speed.
- ✓ If we go through the algorithm for 8 queens 981 queen moves (876 position tests plus 105 backtracks) are required for the first solution alone. 16,704 moves (14,852 tests and 1852 backtracks) are needed to find all 92 solutions.
- ✓ Given those figures, it's easy to see why the solution is best left to computers.

THANK YOU

Graph Searching/Traversal: Breadth First Search & Depth First Search

Ajit Kumar Behera

Graph terminology - overview

- A **graph** $G = (V, E)$ consists of two sets
 - set of **vertices** $V = \{v_1, v_2, \dots, v_n\}$
 - set of **edges** that connect the vertices $E = \{e_1, e_2, \dots, e_m\}$
- An edge $e = (u, v)$ is a pair of vertices u and v and is said to be incident with u and v .

Graph terminology - overview

- Two vertices in a graph are **adjacent** if there is an edge connecting the vertices.
- A graph is termed as **weighted graph** if all the edges in it are labeled with some weights.
- Two vertices are on a **path** if there is a sequences of vertices beginning with the first one and ending with the second one.
- If there is an edge start and end with same vertices it is called **self loop**

Graph terminology - overview

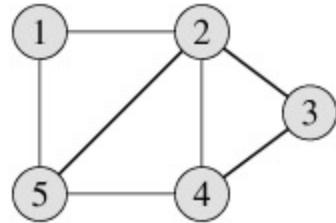
- A graph G is a **directed graph** if, for $G=(V, E)$
 - V is the set of vertices
 - E is the set of ordered pair of elements from V
- **Degree of a vertex**
 - No. of edges connected with v_i
 - For directed graphs, vertices have **in** and **out degrees**.
 - $\text{Indegree}(v_i) =$ no. of edges incident in to v_i
 - $\text{Outdegree}(v_i) =$ no. of edges emanating from v_i
- **Connected graph-** If there is path from vertices v_i and v_j

Graph representation – undirected

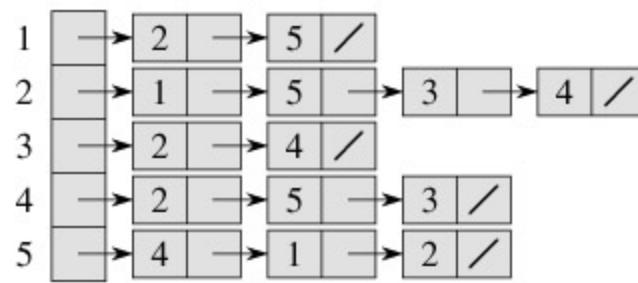
Two methods of representation:

- ❑ **Matrix representation** – Graph can be represented as matrices
 - ❑ **Adjacency matrix**
 - Adjacency matrix A for G with n vertices is an $n \times n$ matrix s.t.
 - $$A_{ij} = \begin{cases} 1, & \text{if there is an edge from } v_i \text{ to } v_j \\ 0, & \text{if there is no such edge} \end{cases}$$
 - ❑ **Incidence matrix**
- ❑ **Linked list representation (Adjacency list)** – Store all vertices in a list and then for each vertex, we have a linked list of its adjacent vertices

Graph representation – undirected



(a)



(b)

	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

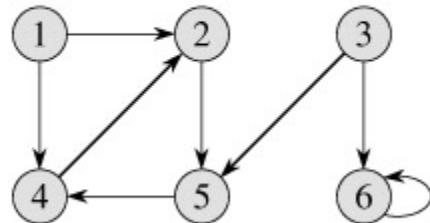
(c)

graph

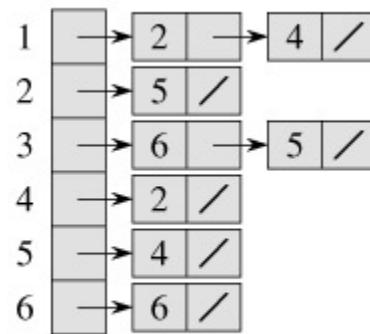
Adjacency list

Adjacency matrix

Graph representation – directed



(a)



(b)

	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

(c)

graph

Adjacency list

Adjacency matrix

Some notes

- Adjacency list: $\Theta(V+E)$
 - Preferred for **sparse** graph
 - $|E|$ is much less than $|V|^2$
 - $\text{Adj}[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$
 - Weighted graph: $w(u, v)$ is stored with vertex v in $\text{Adj}[u]$
 - No quick way to determine if a given edge is present in the graph
- Adjacency matrix: $\Theta(V^2)$
 - Preferred for **dense** graph
 - Symmetry for undirected graph
 - Weighted graph: store $w(u, v)$ in the (u, v) entry
 - Easy to determine if a given edge is present in the graph

Graph-searching/Traversal Algorithms

- Searching a graph means systematically following the edges of the graph so as to visit the vertices.
- Standard graph-searching algorithms.
 - Breadth-first Search (BFS)
 - Queue data structure is used in implementing BFS
 - Depth-first Search (DFS)
 - Stack data structure is used in implementing DFS.

Breadth first search

- Given
 - a graph $G=(V,E)$
 - a distinguished source vertex s
- Breadth first search systematically explores the edges of G to discover every vertex that is reachable from s .
- It also produces a '**breadth first tree**' with root s that contains all the vertices reachable from s .
- For any vertex v **reachable** from s , the path in the breadth first tree corresponds to the shortest path in graph G from s to v .
- It works on both directed and undirected graphs.

Breadth first search

It is so named because

It discovers all vertices at distance k from s before discovering vertices at distance $k+1$.

Animation:

http://en.wikipedia.org/wiki/Image:Animated_BFS.gif

Breadth first search - concepts

- To keep track of progress, it colors each vertex - white, gray or black.
 - **White** – Undiscovered.
 - **Gray** – Discovered but not finished.
 - **Black** – Finished.
 - All vertices starts with **white** and may later become gray then black
 - Colors are required only to reason about the algorithm. Can be implemented without colors.

Breadth first search - concepts

- If $(u,v) \in E$ and vertex u is black, then vertex v is either gray or black
 - i.e. All vertices adjacent to black vertices have been discovered
 - Gray vertices may have some adjacent white vertices

BFS – How it produces a Breadth first tree

- The tree initially contains only root. – s
- Whenever a white vertex v is discovered in scanning adjacency list of vertex u
 - Vertex v and edge (u,v) are added to the tree.
 - Color of each vertex $u \in V$ is stored in $\text{color}[u]$
 - Predecessor of u is stored in variable $\pi[u]$
 - Distance from source s to vertex u is stored in $d[v]$

BFS(G , s)

```
1   for each vertex  $u \in V[G] - \{s\}$ 
2       do  $color[u] \leftarrow \text{WHITE}$ 
3            $d[u] \leftarrow \infty$ 
4            $\pi[u] \leftarrow \text{NIL}$ 
5    $color[s] \leftarrow \text{GRAY}$ 
6    $d[s] \leftarrow 0$ 
7    $\pi[s] \leftarrow \text{NIL}$ 
8    $Q \leftarrow \emptyset$ 
9   ENQUEUE( $Q$ ,  $s$ )
10  while  $Q \neq \emptyset$ 
11      do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12          for each  $v \in Adj[u]$ 
13              do if  $color[v] = \text{WHITE}$ 
14                  then  $color[v] \leftarrow \text{GRAY}$ 
15                   $d[v] \leftarrow d[u] + 1$ 
16                   $\pi[v] \leftarrow u$ 
17                  ENQUEUE( $Q$ ,  $v$ )
18       $color[u] \leftarrow \text{BLACK}$ 
```

Initialization

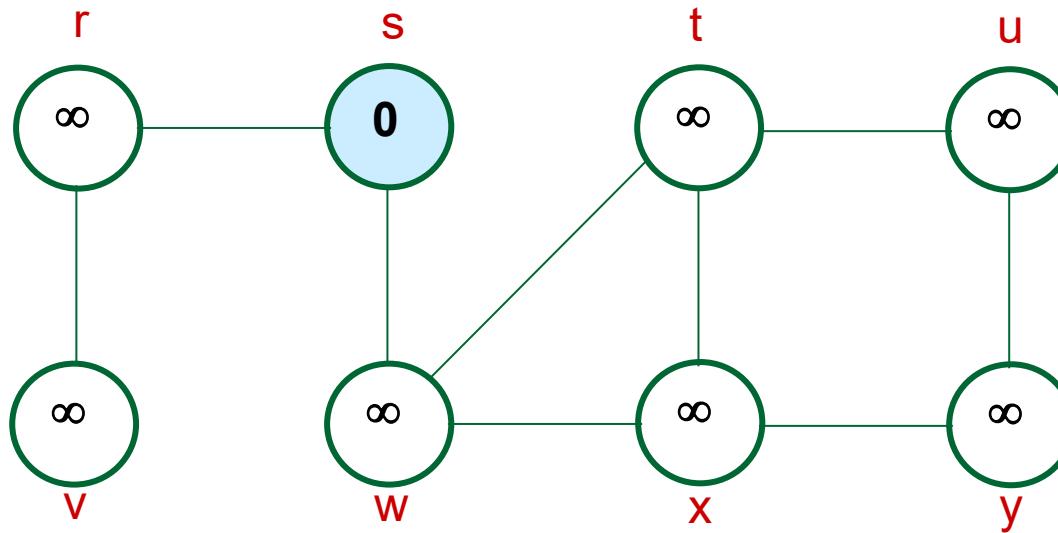
Set up s and initialize Q

Explore all the vertices adjacent to u and update d , π and Q

BFS - algorithm

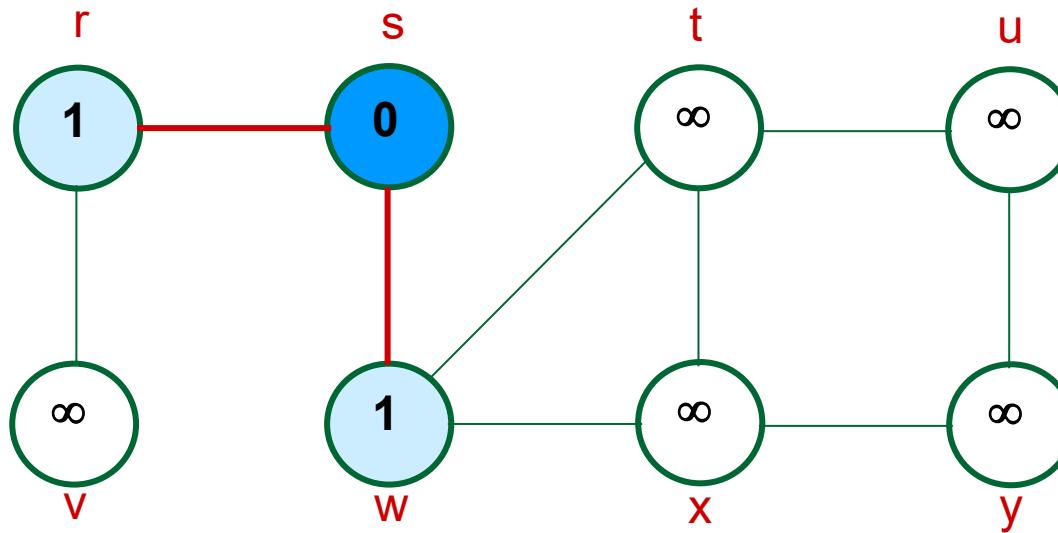
```
BFS(G, s)                                // G is the graph and s is the starting node
1 for each vertex u ∈ V [G] - {s}
2   do color[u] ← WHITE      // color of vertex u
3   d[u] ← ∞                  // distance from source s to vertex u
4   π[u] ← NIL                // predecessor of u
5 color[s] ← GRAY
6 d[s] ← 0
7 π[s] ← NIL
8 Q ← ∅                                // Q is a FIFO - queue
9 ENQUEUE(Q, s)
10 while Q ≠ ∅                         // iterates as long as there are gray vertices. Lines 10-18
11   do u ← DEQUEUE(Q)
12     for each v ∈ Adj[u]
13       do if color[v] = WHITE          // discover the undiscovered adjacent vertices
14         then color[v] ← GRAY        // enqueued whenever painted gray
15         d[v] ← d[u] + 1
16         π[v] ← u
17         ENQUEUE(Q, v)
18   color[u] ← BLACK      // painted black whenever dequeued
```

Example (BFS)



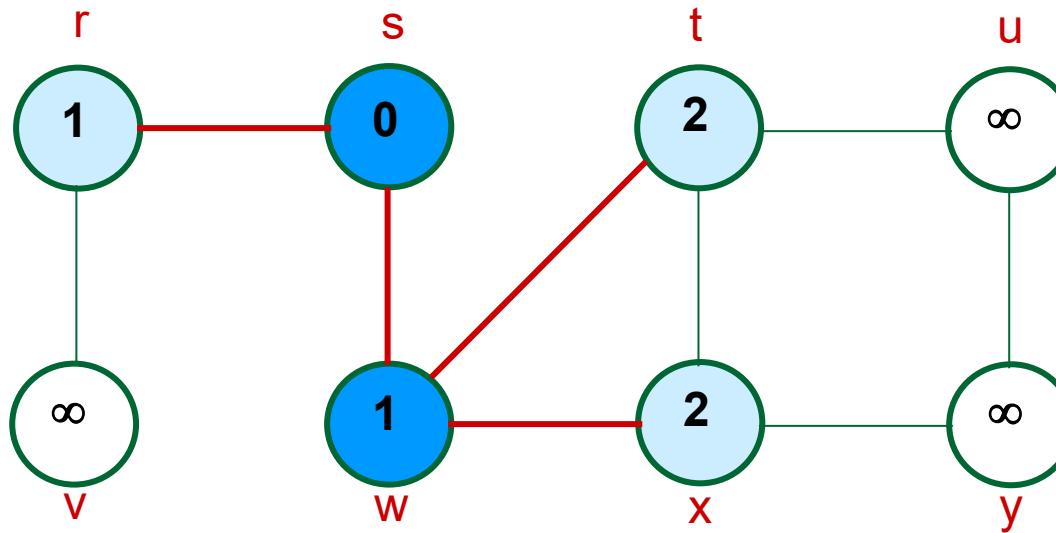
Q: s
0

Example (BFS)



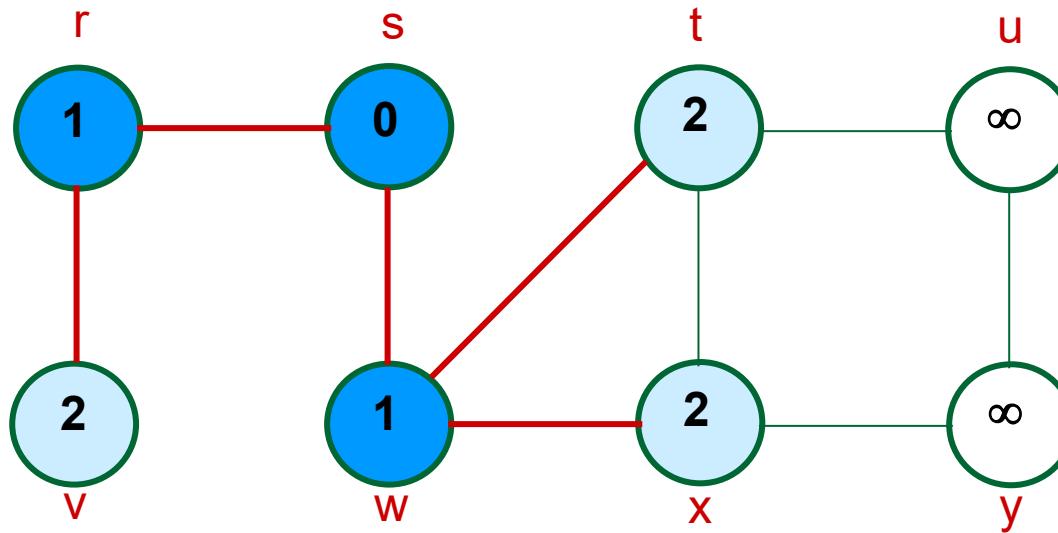
Q:	w	r
	1	1

Example (BFS)



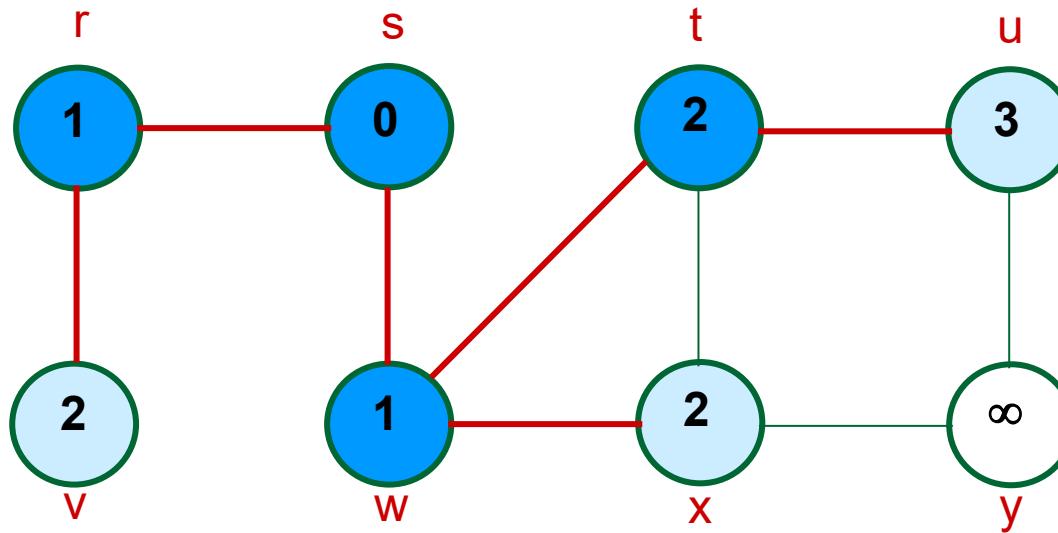
Q:	r	t	x
	1	2	2

Example (BFS)



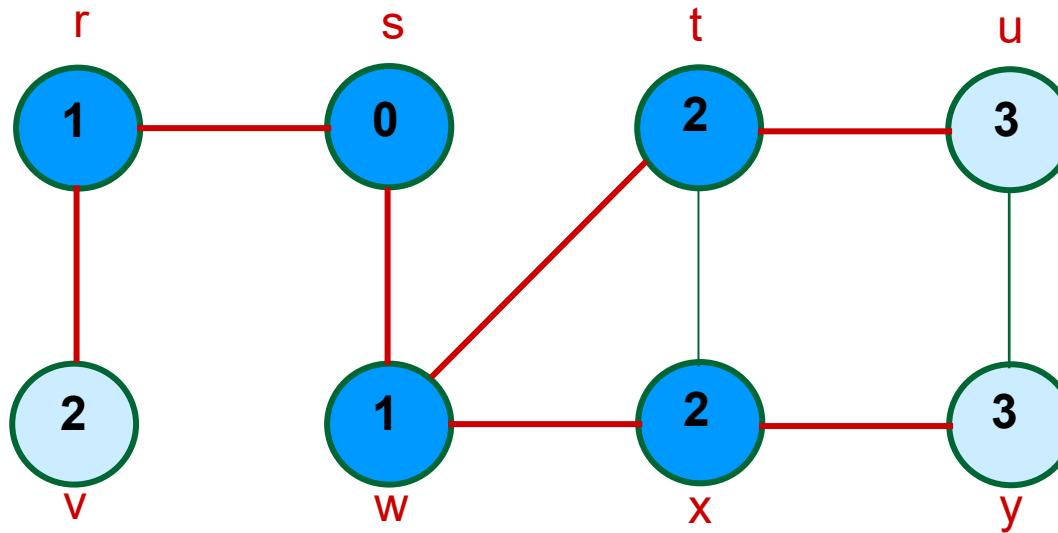
Q:	t	x	v
	2	2	2

Example (BFS)



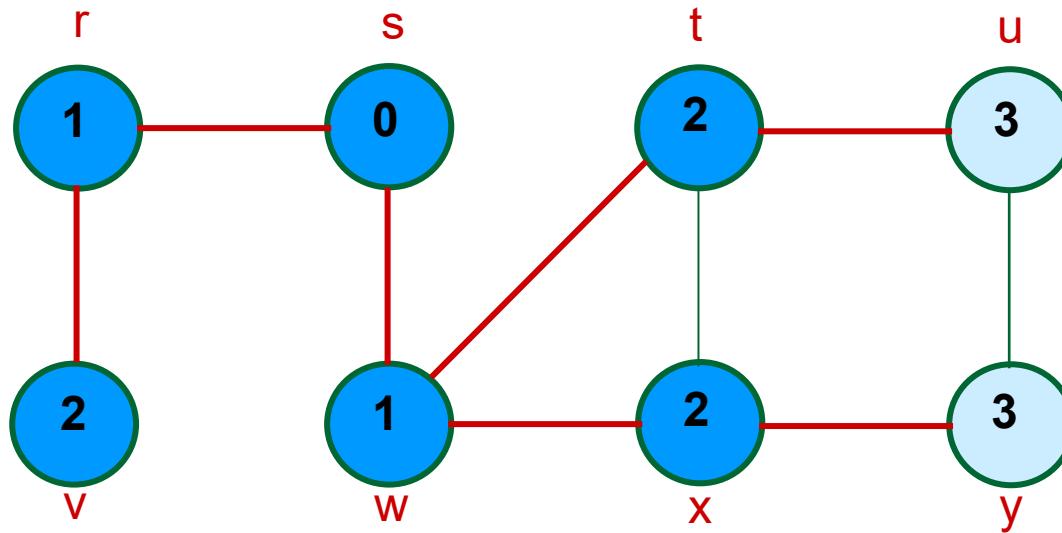
Q:	x	v	u
	2	2	3

Example (BFS)



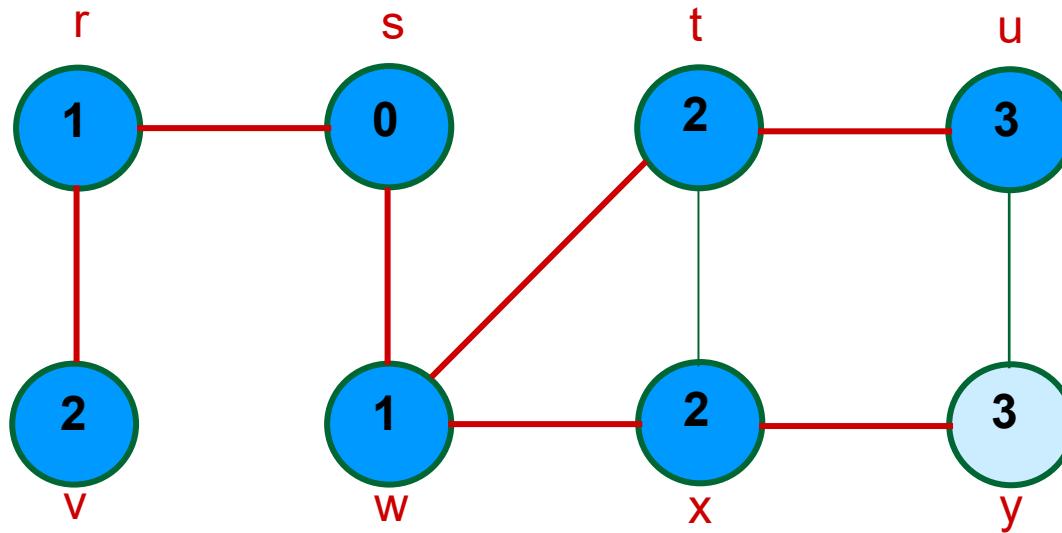
Q:	v	u	y
	2	3	3

Example (BFS)



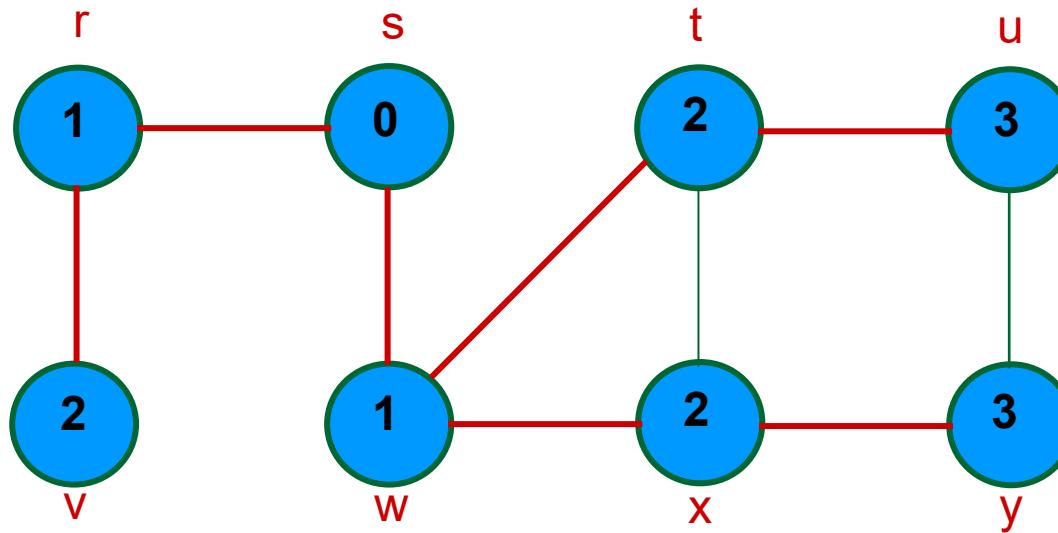
Q:	u	y
	3	3

Example (BFS)



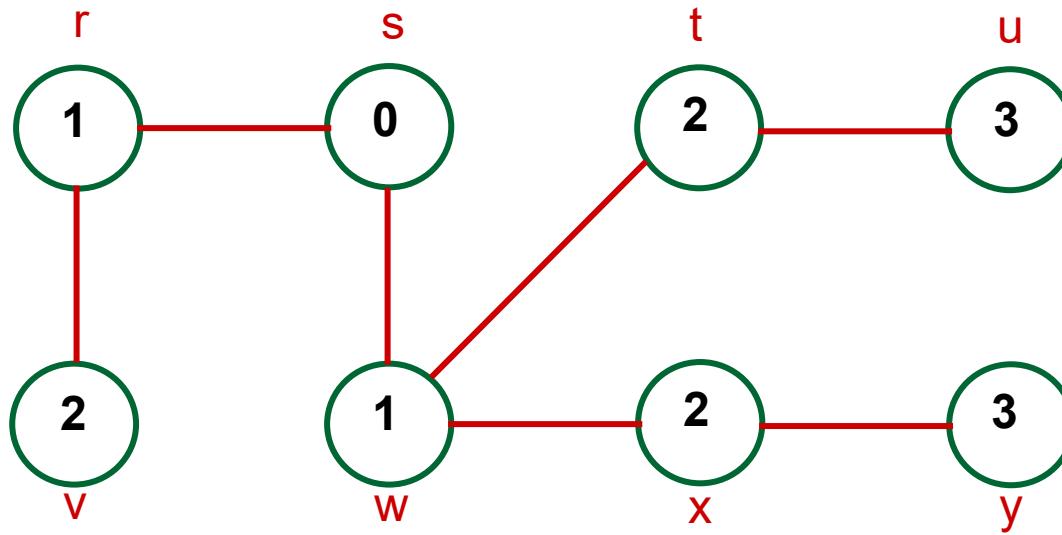
Q: y
3

Example (BFS)



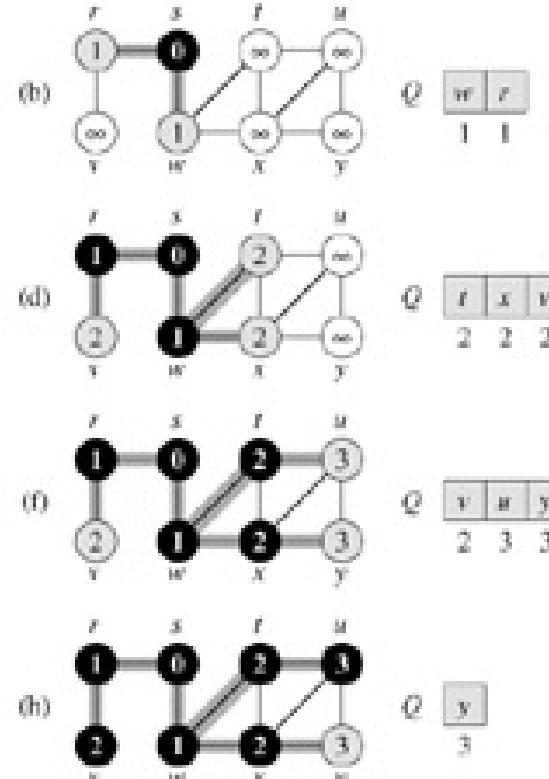
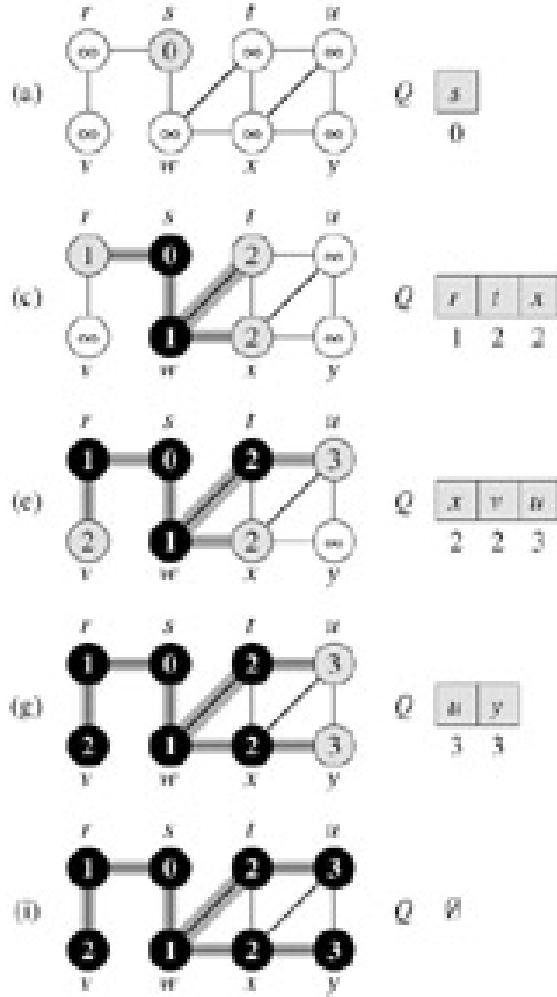
Q: \emptyset

Example (BFS)



BF Tree

Breadth First Search - example



Breadth first search - analysis

- $O(V+E)$
 - Each vertex is en-queued ($O(1)$) at most once $\rightarrow O(V)$
 - Each adjacency list is scanned at most once $\rightarrow O(E)$
 - Initialization overhead $\rightarrow O(V)$

Total runtime $O(V+E)$

BFS – printing the path

Print-Path(G, s, v)

1. **if** $v == s$
2. **then** print s
3. **else if** $\pi[v] == \text{NIL}$
4. **then** print “No path from s to v exists”
5. **else** Print-Path($G, s, \pi[v]$)
6. print v

Depth first search

- “Search as deep as possible first.”
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- When all of v ’s edges have been explored, the search “**backtracks**” to explore edges leaving the vertex from which v was discovered.
- Process continues until we have discovered all the vertices that are **reachable** from the original source.

Depth first search

- Vertices go through white, gray and black stages of color.
 - White – initially
 - Gray – when discovered first
 - Black – when finished
 - i.e. the adjacency list of the vertex is completely examined.
- Also records *timesteps* for each vertex
 - $d[v]$ - when the vertex is first discovered
 - $f[v]$ - when the vertex is finished

Depth first search

- DFS will create a **forest of DFS-trees**
 - If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source
 - The predecessor subgraph of a DFS forms a *depth-first forest* composed of several *depth-first trees*.
 - The entire process is repeated until all vertices are discovered

Depth first search - algorithm

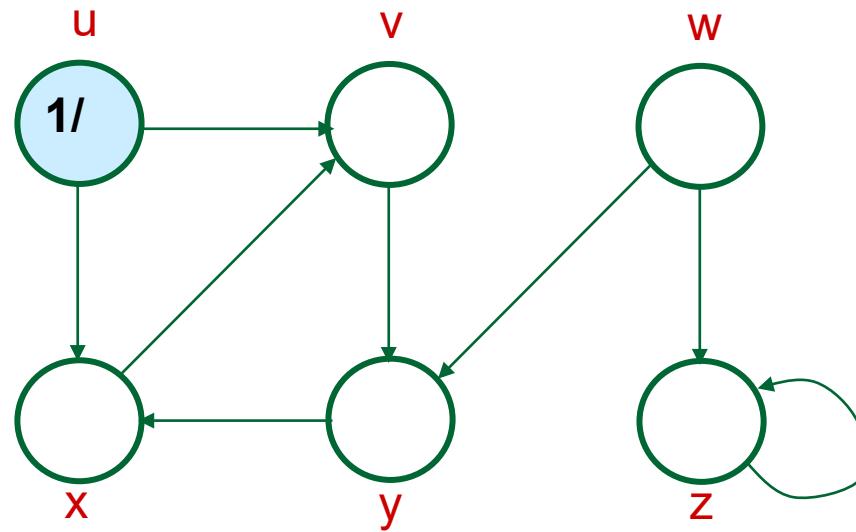
DFS(G)

```
1 for each vertex  $u \in V[G]$            // color all vertices white, set their parents NIL
2   do  $\text{color}[u] \leftarrow \text{WHITE}$ 
3    $\pi[u] \leftarrow \text{NIL}$ 
4 time  $\leftarrow 0$                          // zero out time
5 for each vertex  $u \in V[G]$            // call only for unexplored vertices
6   do if  $\text{color}[u] == \text{WHITE}$        // this may result in multiple sources
7     then DFS-VISIT( $u$ )
```

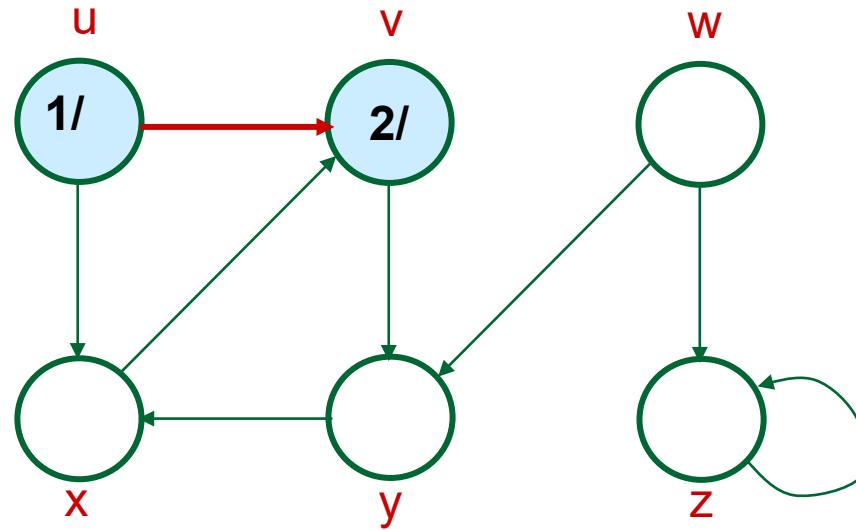
DFS-VISIT(u)

```
1  $\text{color}[u] \leftarrow \text{GRAY}$           ▷ White vertex  $u$  has just been discovered.
2 time  $\leftarrow \text{time} + 1$ 
3  $d[u] = \text{time}$                       // record the discovery time
4 for each  $v \in \text{Adj}[u]$             ▷ Explore edge( $u, v$ ).
5   do if  $\text{color}[v] == \text{WHITE}$ 
6     then  $\pi[v] \leftarrow u$               // set the parent value
7     DFS-VISIT( $v$ )                  // recursive call
8  $\text{color}[u] = \text{BLACK}$              ▷ Blacken  $u$ ; it is finished.
9  $f[u] = \text{time} \leftarrow \text{time} + 1$ 
```

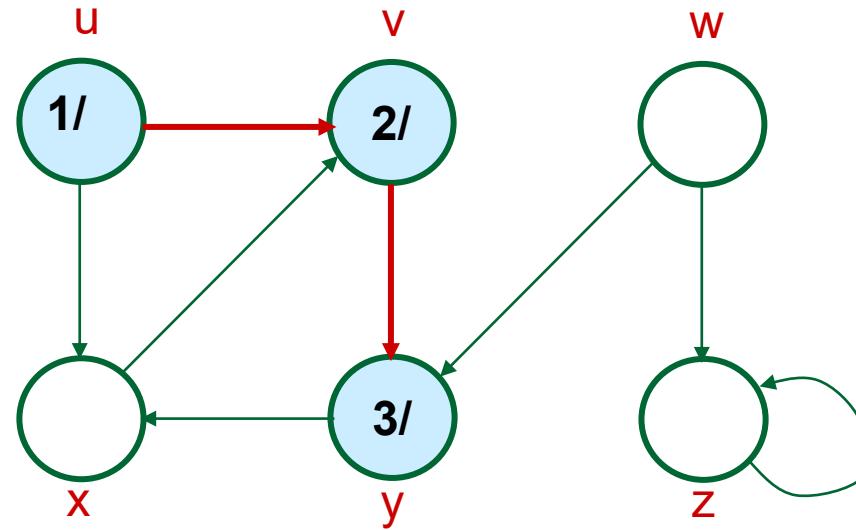
Example (DFS)



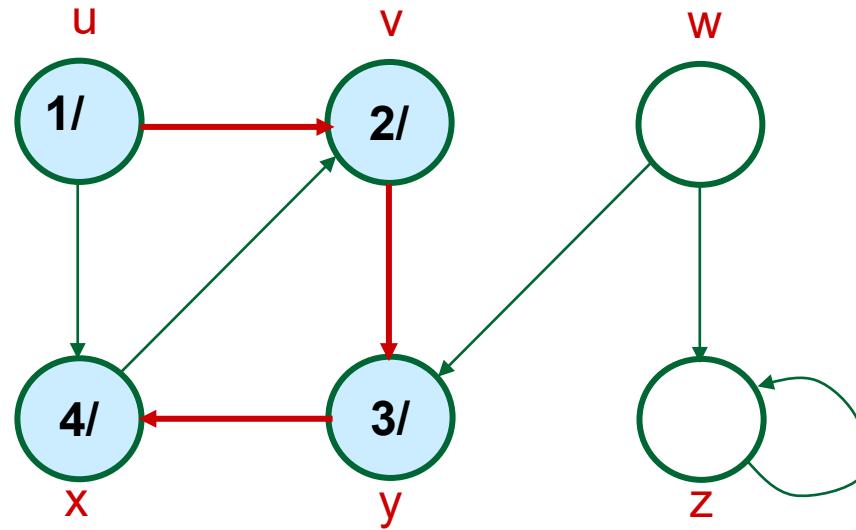
Example (DFS)



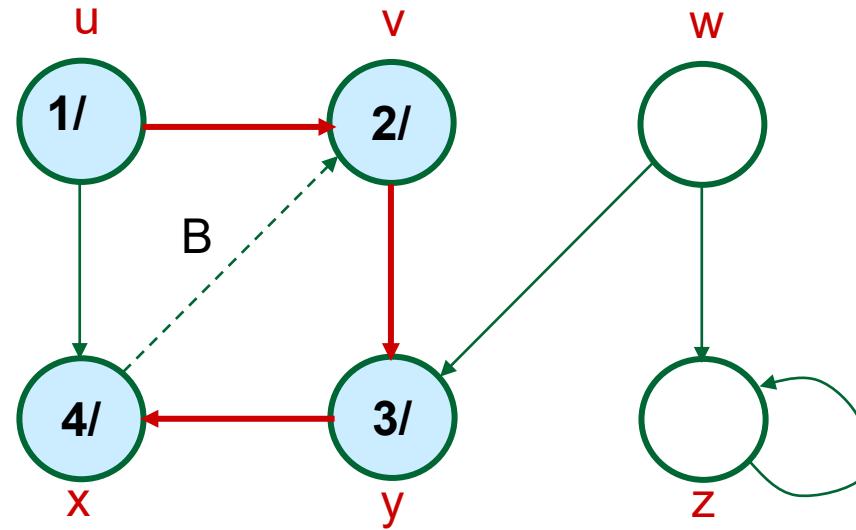
Example (DFS)



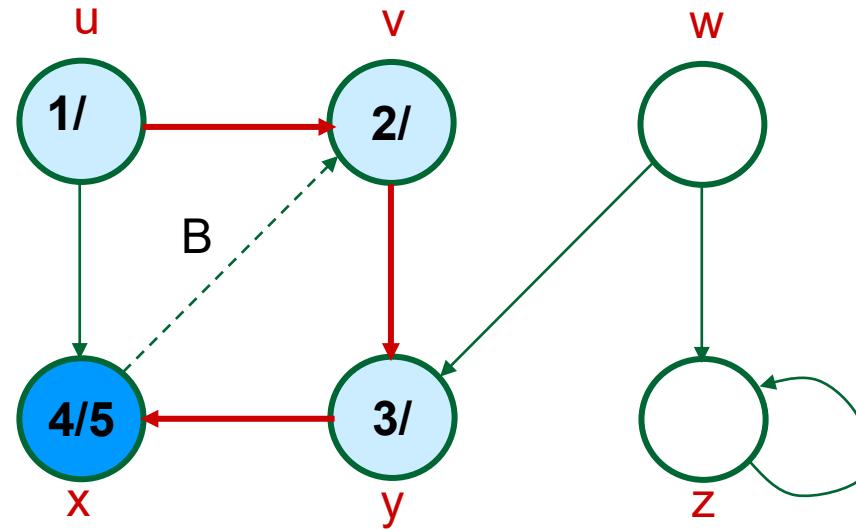
Example (DFS)



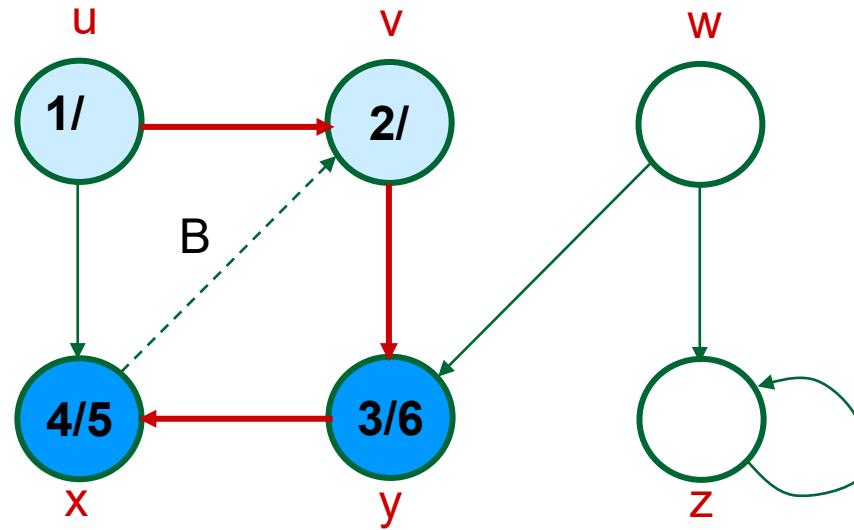
Example (DFS)



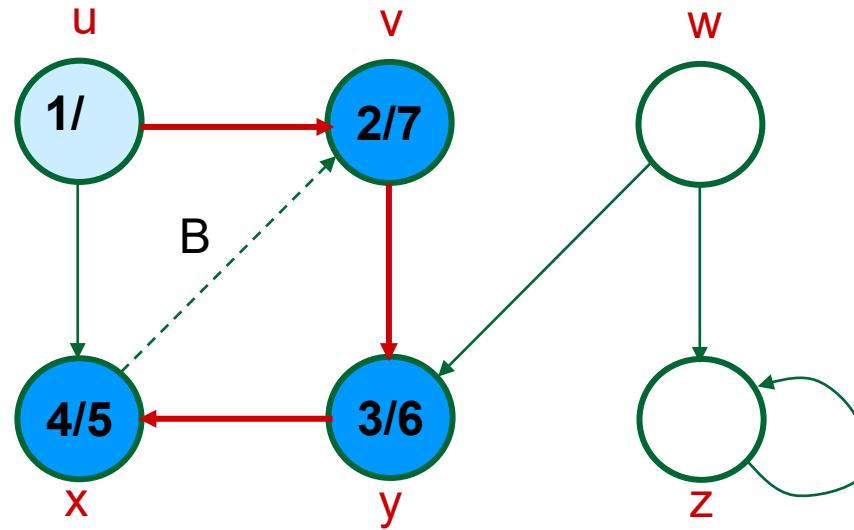
Example (DFS)



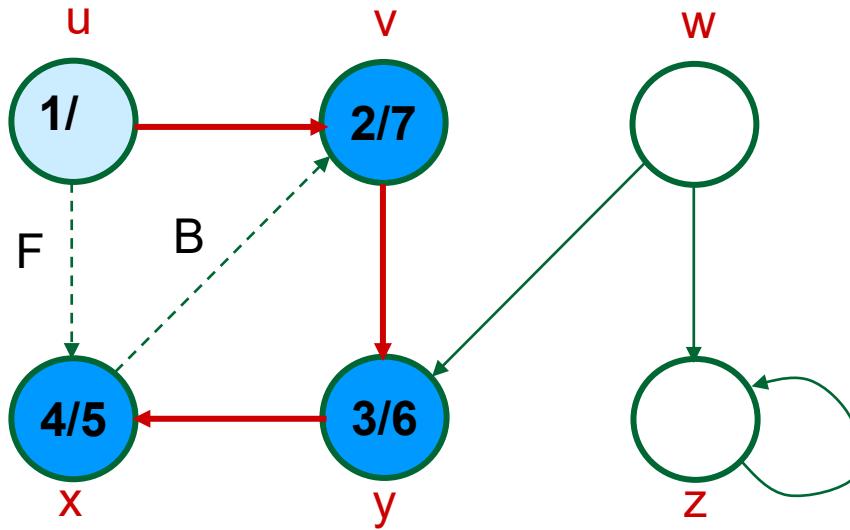
Example (DFS)



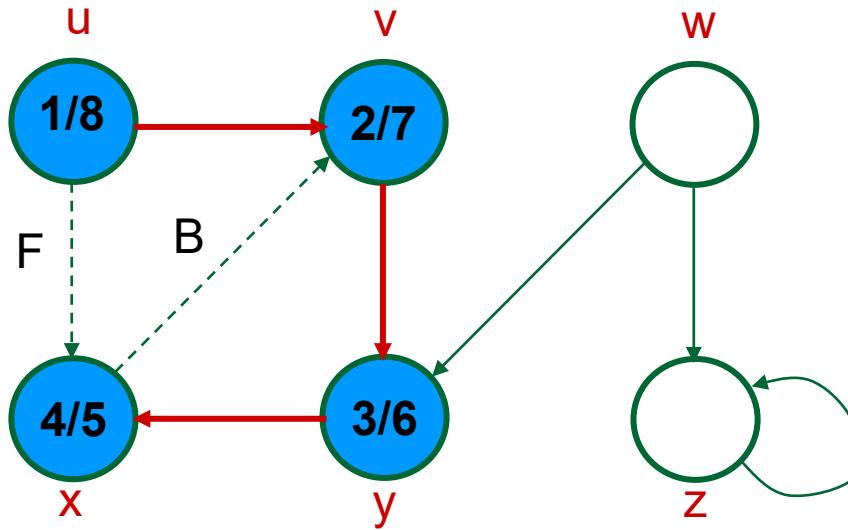
Example (DFS)



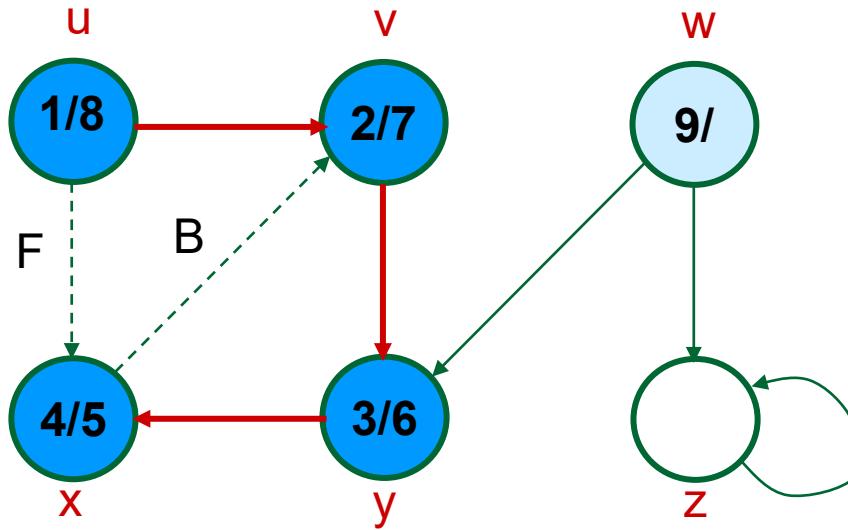
Example (DFS)



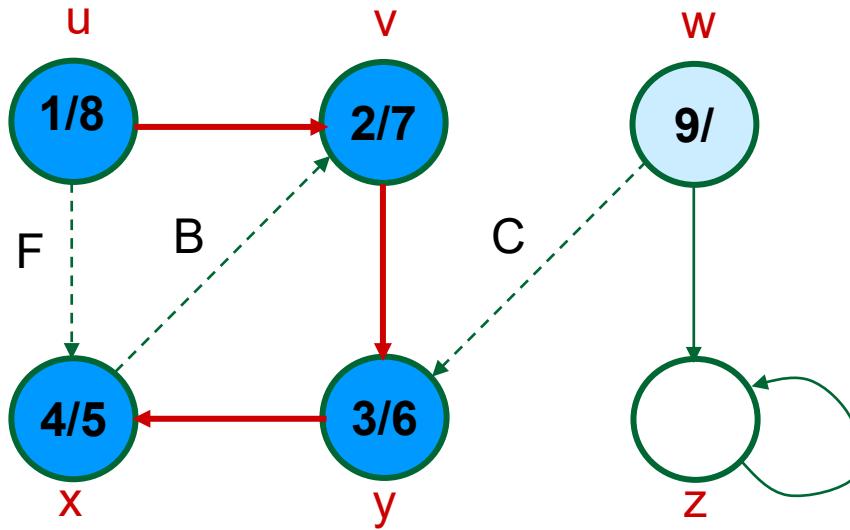
Example (DFS)



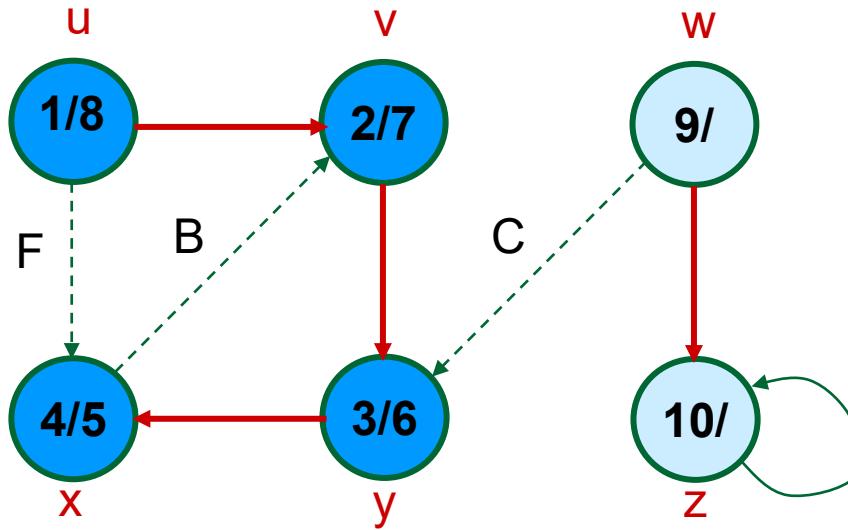
Example (DFS)



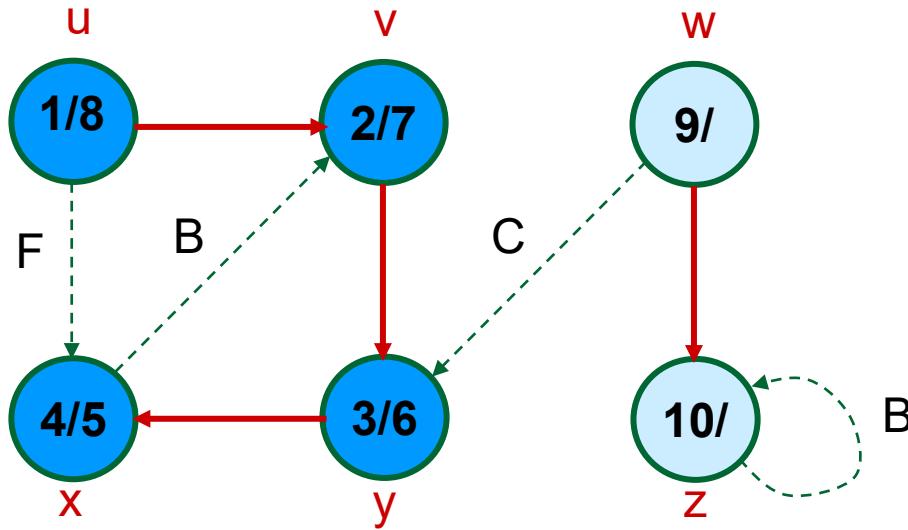
Example (DFS)



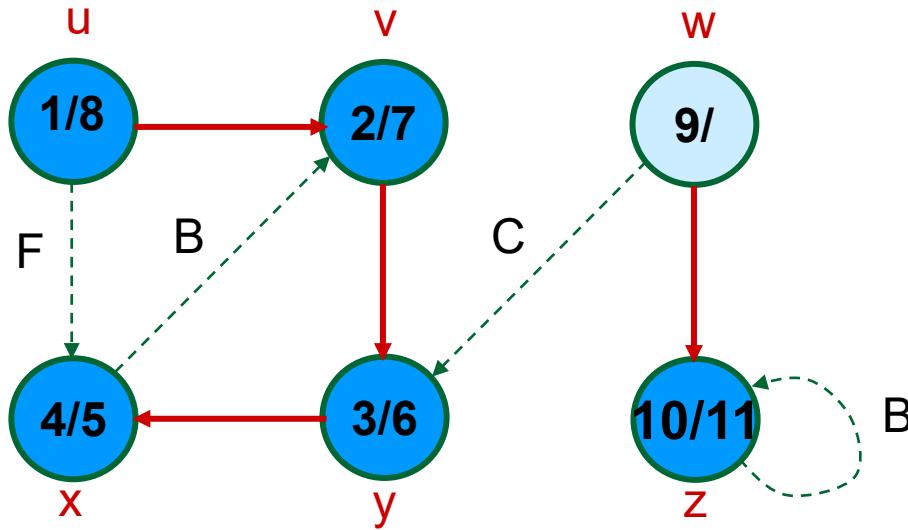
Example (DFS)



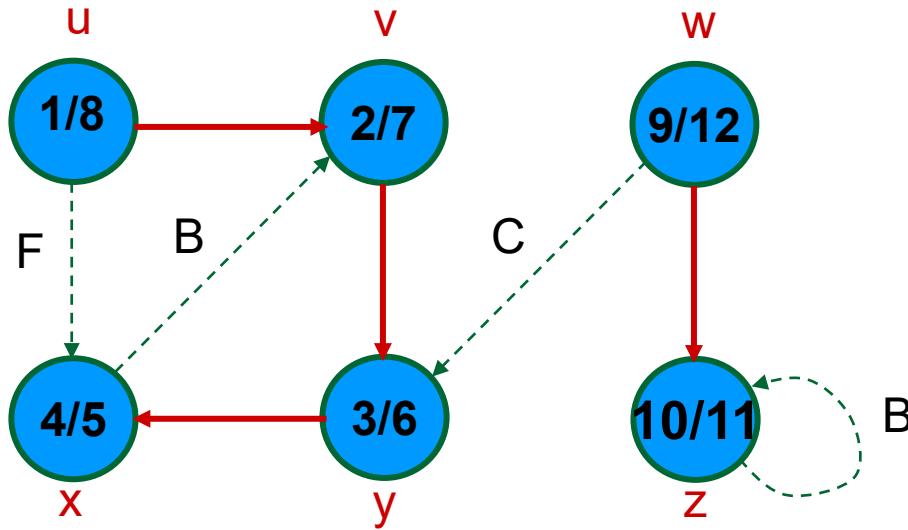
Example (DFS)



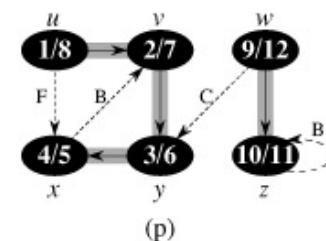
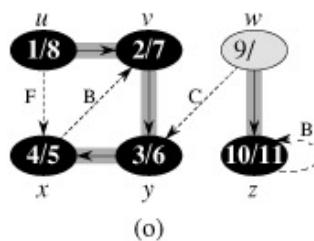
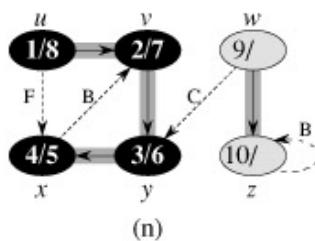
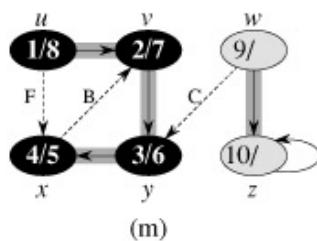
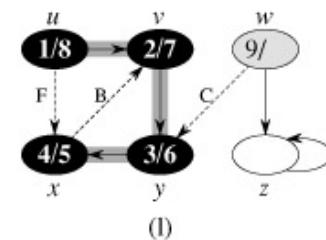
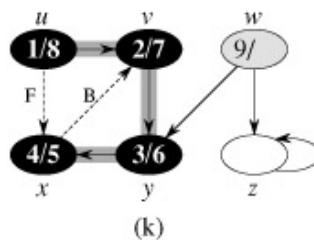
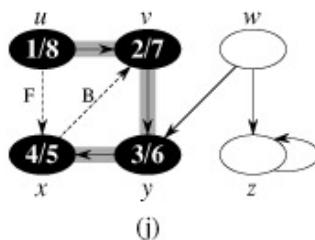
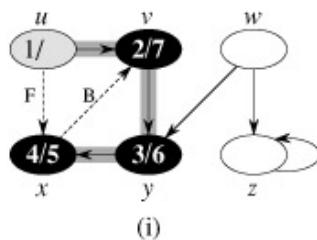
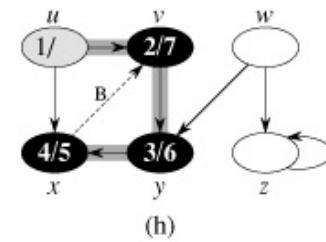
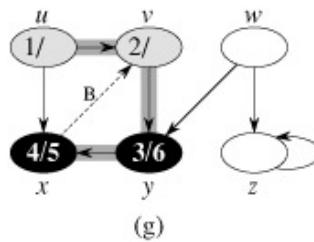
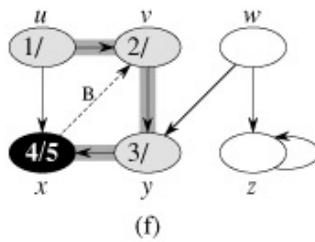
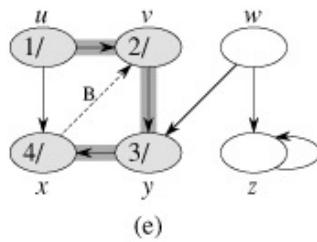
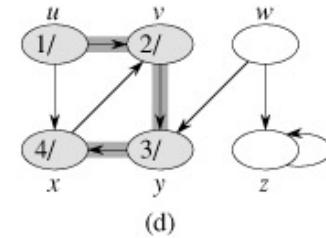
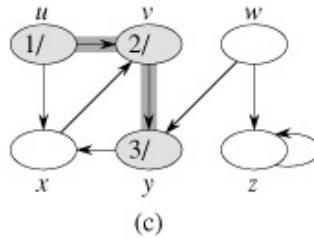
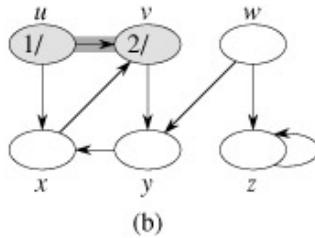
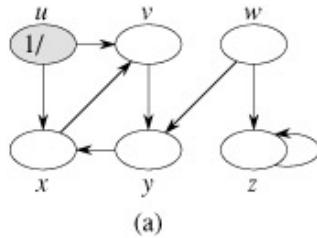
Example (DFS)



Example (DFS)



Depth first search – example



Depth first search - analysis

- Lines 1-3, initialization take time $\Theta(V)$.
- Lines 5-7 take time $\Theta(V)$, excluding the time to call the DFS-VISIT.
- DFS-VISIT is called only once for each node (since it's called only for white nodes and the first step in it is to paint the node gray).
- Loop on line 4-7 is executed $|Adj(v)|$ times. Since, $\sum_{v \in V} |Adj(v)| = \Theta(E)$, the total cost of DFS-VISIT is $\Theta(E)$

The total cost of DFS is $\Theta(V+E)$

BFS and DFS - comparison

- Space complexity of DFS is lower than that of BFS.
- Time complexity of both is same – $O(|V|+|E|)$.
- The behavior differs for graphs where not all the vertices can be reached from the given vertex s .
- Predecessor subgraphs produced by DFS may be different than those produced by BFS. The BFS product is just one tree whereas the DFS product may be multiple trees.

BFS and DFS – possible applications

- Exploration algorithms in Artificial Intelligence
- Possible to use in routing / exploration wherever travel is involved. E.g.,
 - I want to explore all the nearest pizza places and want to go to the nearest one with only two intersections.
 - Find distance from a factory to every delivery center.
 - Most of the mapping software (GOOGLE maps, YAHOO(?) maps) should be using these algorithms.
- Applications of DFS
 - Topologically sorting a directed acyclic graph.
 - List the graph elements in such an order that all the nodes are listed before nodes to which they have outgoing edges.
 - Finding the strongly connected components of a directed graph.
 - List all the subgraphs of a strongly connected graph which themselves are strongly connected.

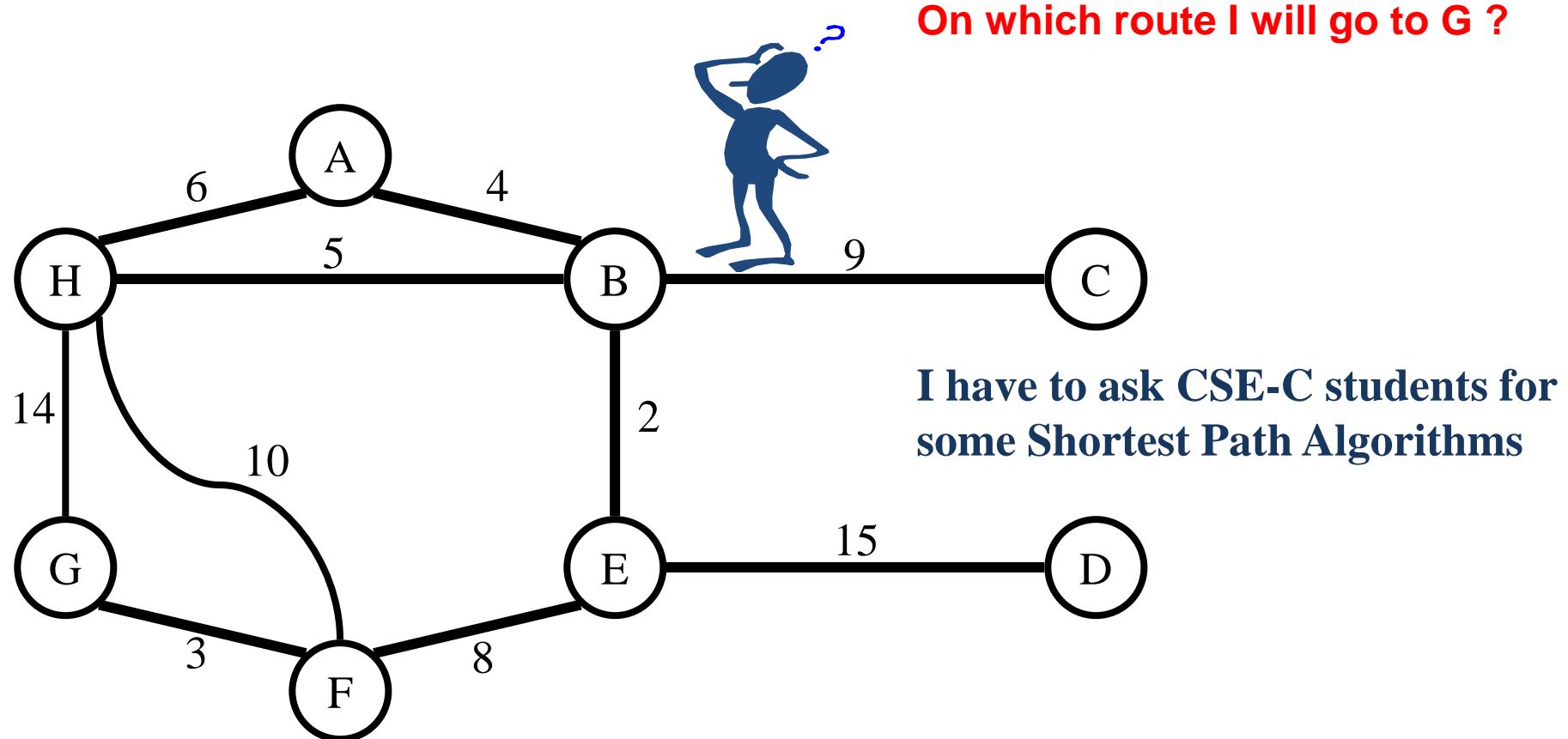
References

- Introduction to Algorithms by Cormen, Thomas et. al., The MIT press.
- [http://en.wikipedia.org/wiki/Graph theory](http://en.wikipedia.org/wiki/Graph_theory)
- [http://en.wikipedia.org/wiki/Depth_first_search](http://en.wikipedia.org/wiki/Depth-first_search)

Shortest Path Algorithms

Ajit Kumar Behera

Shortest Path



Shortest Path

- Shortest path = a path of the minimum weight
- Applications of Shortest Path Algo.
 - static/dynamic network routing
 - robot motion planning
 - route generation in traffic
 - road map applications

Types of Shortest Path Problems

For a Graph $G(V, E)$

Single-Source Shortest Path:

Find a shortest path from a given source (vertex $s \in V$) to all of the vertices $v \in V$.

(One source, Many Destinations)

Single-Destination Shortest Path:

Find shortest path to a given destination vertex

(Many sources, One Destination)

Single-Pair Shortest Path:

Find shortest path from u to v .

(One Source, One Destination)

All-Pair Shortest Path:

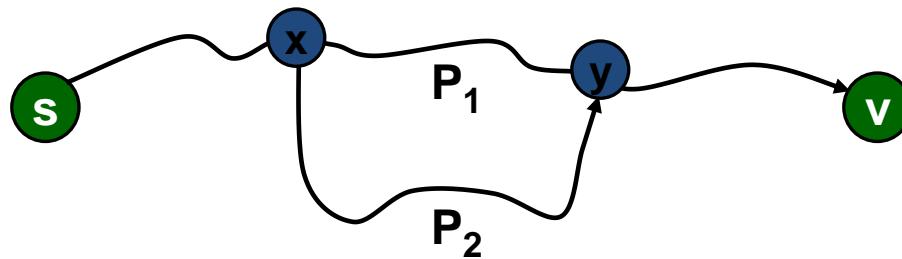
Find Shortest path from u to v , for all $u, v \in V$.

(Many Sources, Many Destinations)

Shortest Path Properties

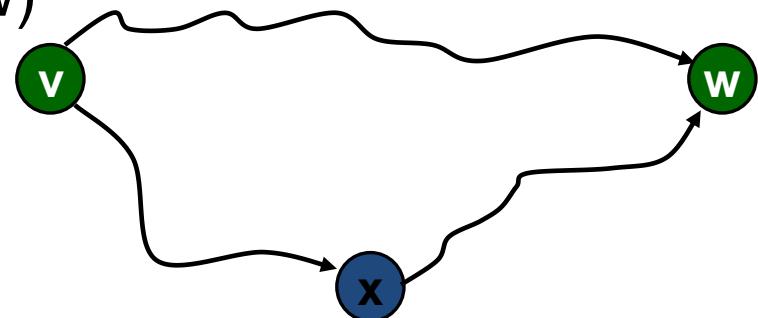
1. Optimal substructure property

All sub-paths of shortest paths are also shortest paths.



2. Triangle inequality.

Let $d(v, w)$ be the length of the shortest path from v to w .
Then, $d(v, w) \leq d(v, x) + d(x, w)$



Basics

Negative Weight and Cycles

If there is a negative weight cycle on some path from s to t

Shortest Path weight can be defined as $\delta(s, t) = -\infty$

Cycle $\langle x, y, x \rangle$ has weight

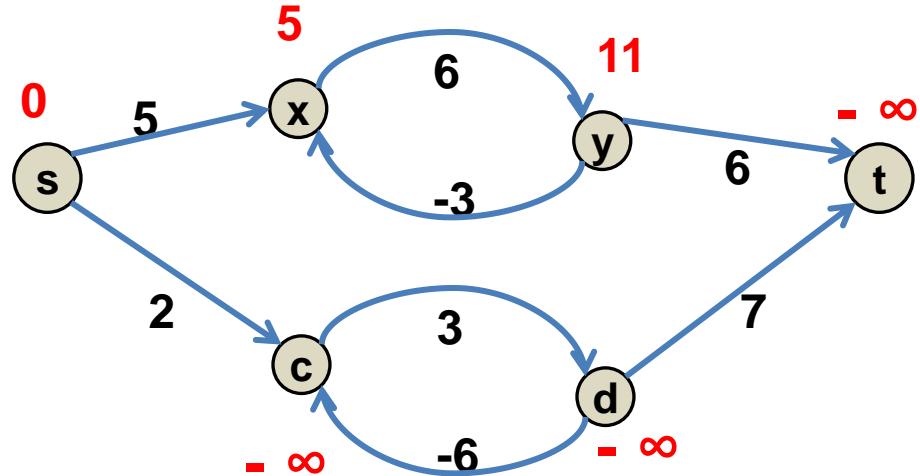
$$6 + (-3) = 3 > 0$$

(positive weight cycle)

Cycle $\langle c, d, c \rangle$ has weight

$$3 + (-6) = -3 < 0$$

(Negative weight cycle)



Shortest Path must not contain Cycles

Negative edges are OK, as long as there are no *negative weight cycles*

Different Shortest Path Algorithms

1. Single-Source Shortest Path Algorithms

1. **Dijkstra's Algorithm:** Doesn't allow negative weight edges
2. **Bellman-ford's Algorithm:** Allows negative weight edges and returns false if there is any negative weight cycles.

2. All-Pair Shortest Path Algorithm

1. **Floyd-Warshall's algorithm:** allows negative weight edges, But assumption is that there are no negative weight cycles

Initialization

All single source shortest path algorithms start with
INITIALIZE-SINGLE-SOURCE routine

```
INITIALIZE-SINGLE-SOURCE (v, s)
```

```
1 for each v ∈ V[G] do
2     d[v] ← ∞
3     π[v] ← NIL
4 d[s] ← 0
```

$d[v]$: shortest Path estimate

$d[v]$: $\delta(s, v)$

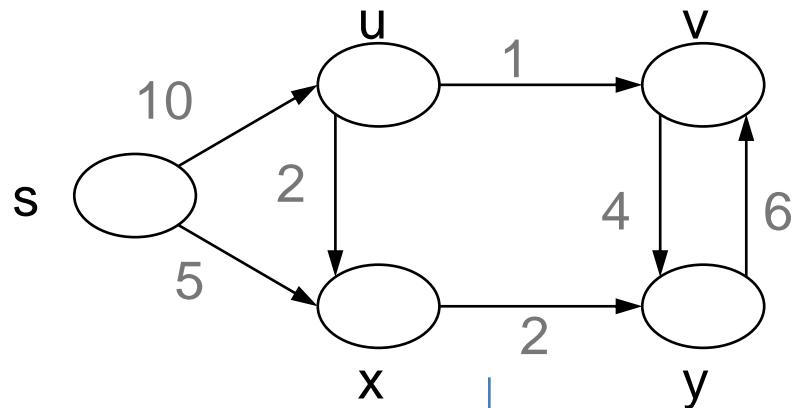
$\pi[v]$: predecessor of v on a shortest path from
source vertex

If no predecessor then $\pi[v] = \text{NIL}$

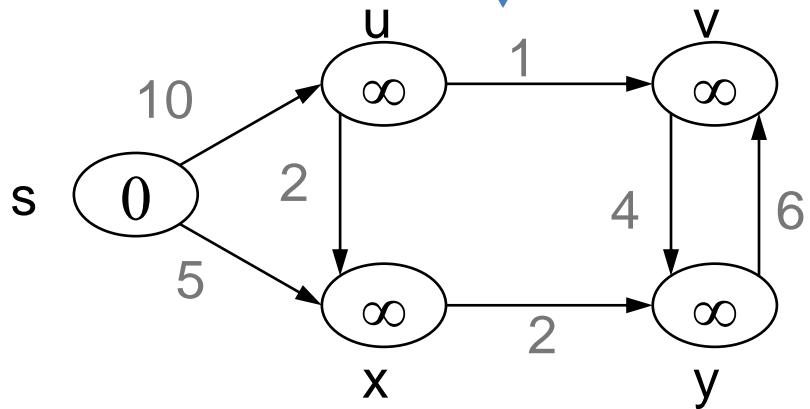
Initialization(Example)

```
INITIALIZE-SINGLE-SOURCE (v, s)
```

```
1 for each v ∈ V[G] do  
2     d[v] ← ∞  
3     π [v] ← NIL  
4 d[s] ← 0
```



```
INITIALIZE-SINGLE-SOURCE (v, s)
```



Edge Relaxation

Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u

Edge Relaxation on edge (u, v) with weight w is as follows

RELAX (u, v, w)

1. if $d[v] > d[u] + w(u, v)$ then
2. $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

Edge Relaxation(Example)

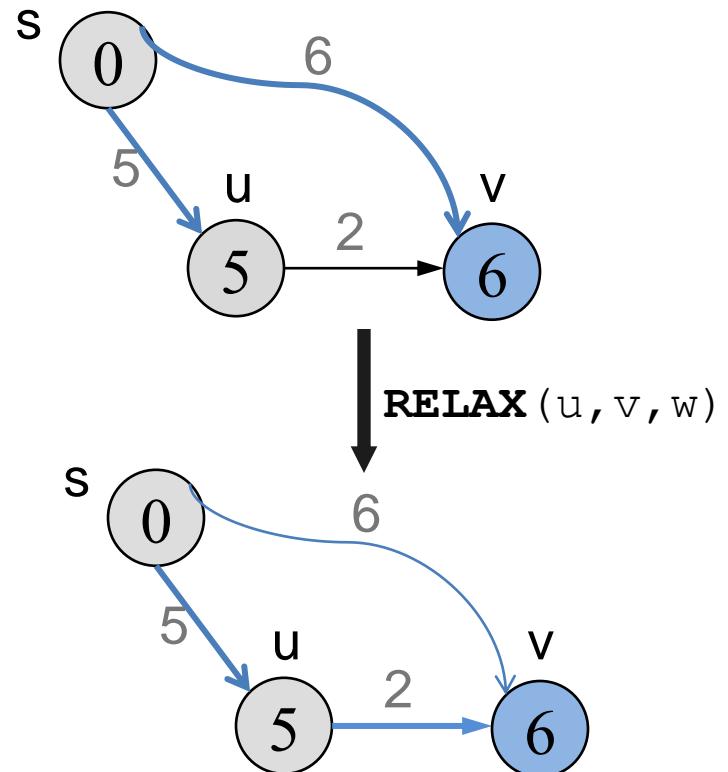
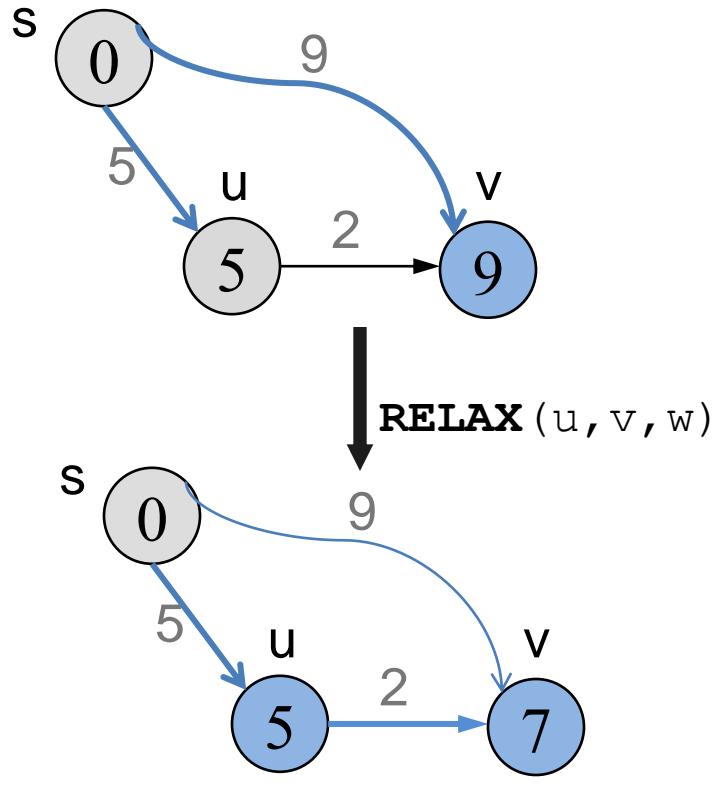
RELAX(u, v, w)

1. if $9 > 5 + 2$ then
2. $d[v] \leftarrow 7$
3. $\pi[v] \leftarrow u$

RELAX(u, v, w)

1. if $6 > 5 + 2$ then

X



Dijkstra's Algorithm

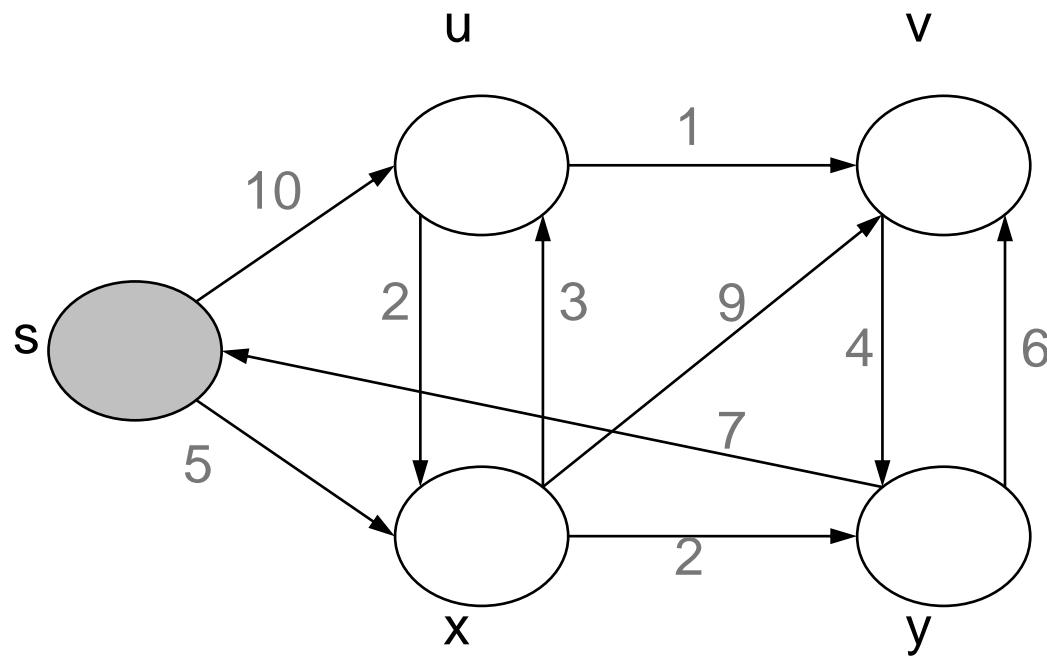
- Solves single-source shortest path problems on a weighted graph.
- Assumption: No. Negative Edge Weights
- Basic Idea
 - S – vertices whose final shortest path weights are determined
 - Q – priority queue = $V - S$
 - At each step select "closest" vertex u , add it to S , and relax all edges from u

Algorithm

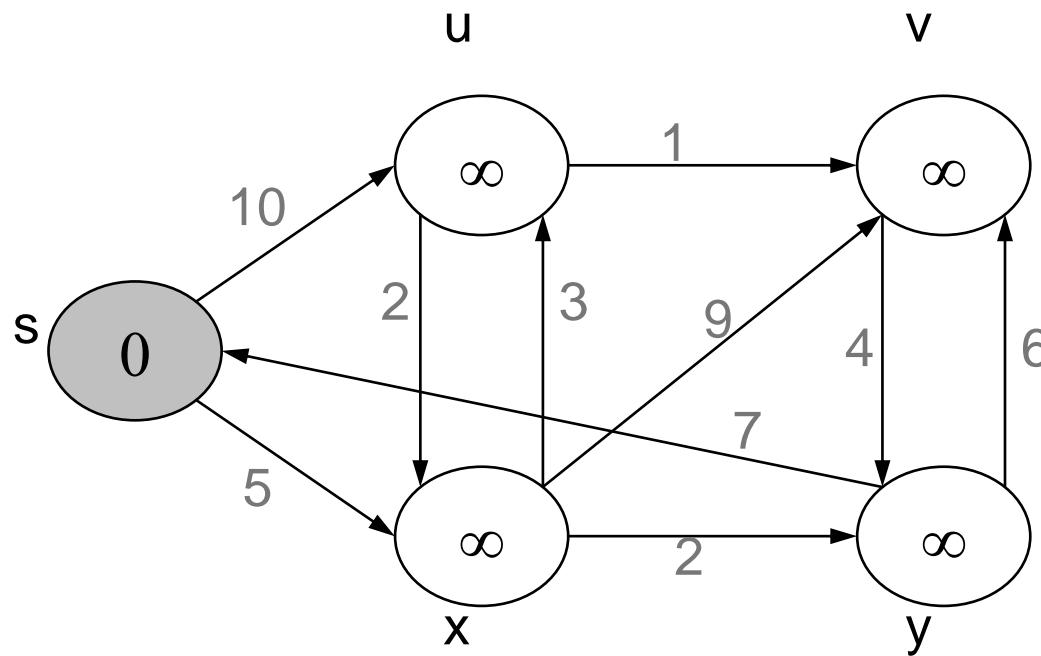
DIJKSTRA(V, E, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. $S \leftarrow \{ \}$
3. $Q \leftarrow V[G]$
4. While $Q \neq \Phi$ do
5. $u \leftarrow \text{EXTRACT-MIN}(Q)$
6. $S \leftarrow S \cup \{u\}$
7. for each vertex $v \in \text{Adj}[u]$ do
8. $\text{RELAX}(u, v, w)$

Dijkstra's Algorithm(Example)



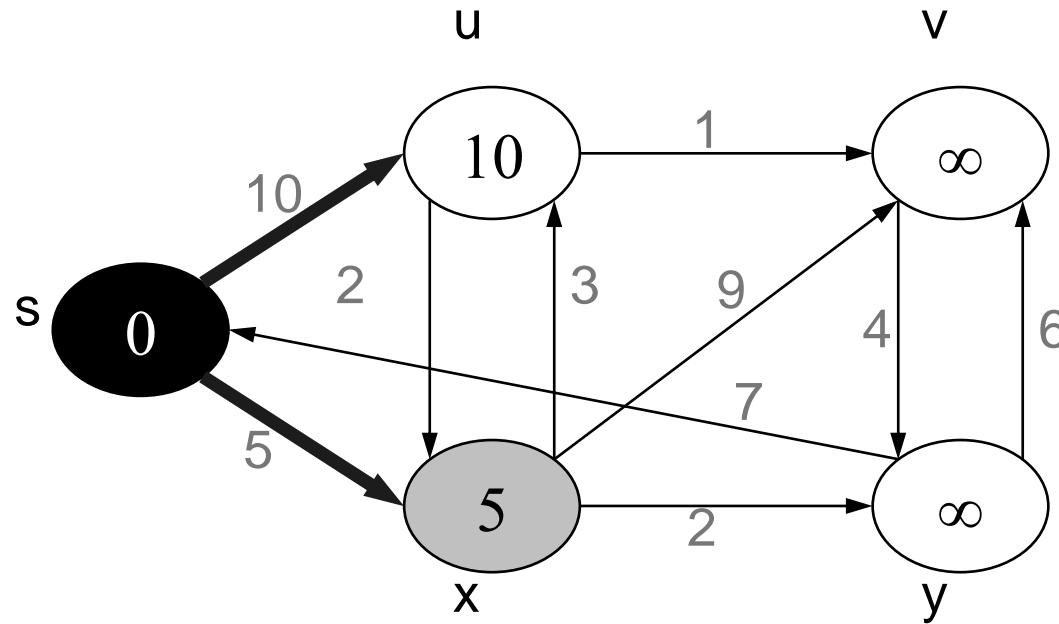
Dijkstra's Algorithm(Example)



$S : \{ \}$

$Q : \{s, u, v, x, y\}$

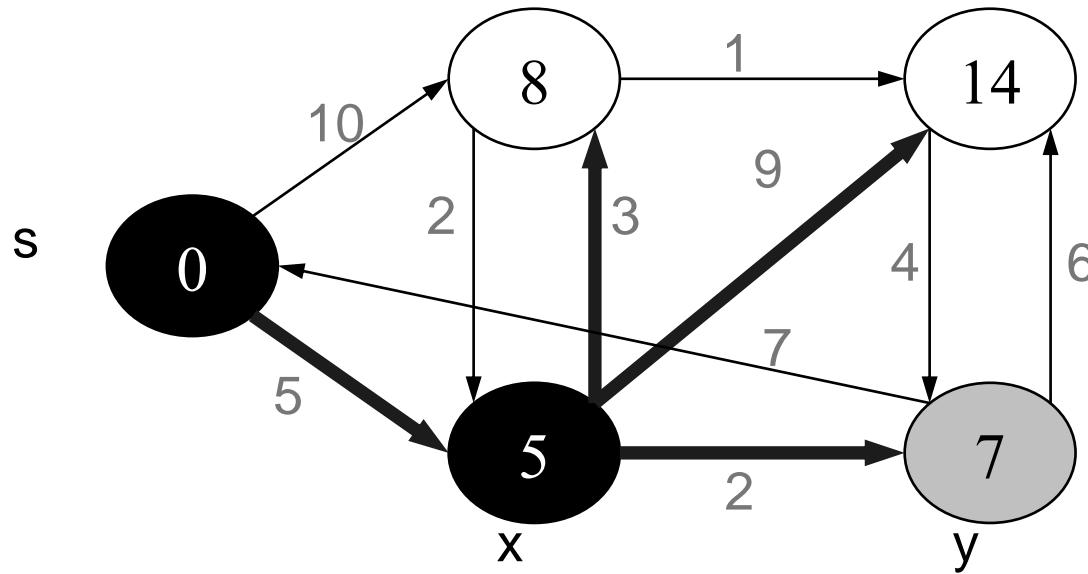
Dijkstra's Algorithm(Example)



$S : \{ s \}$

$Q : \{x, u, v, y\}$

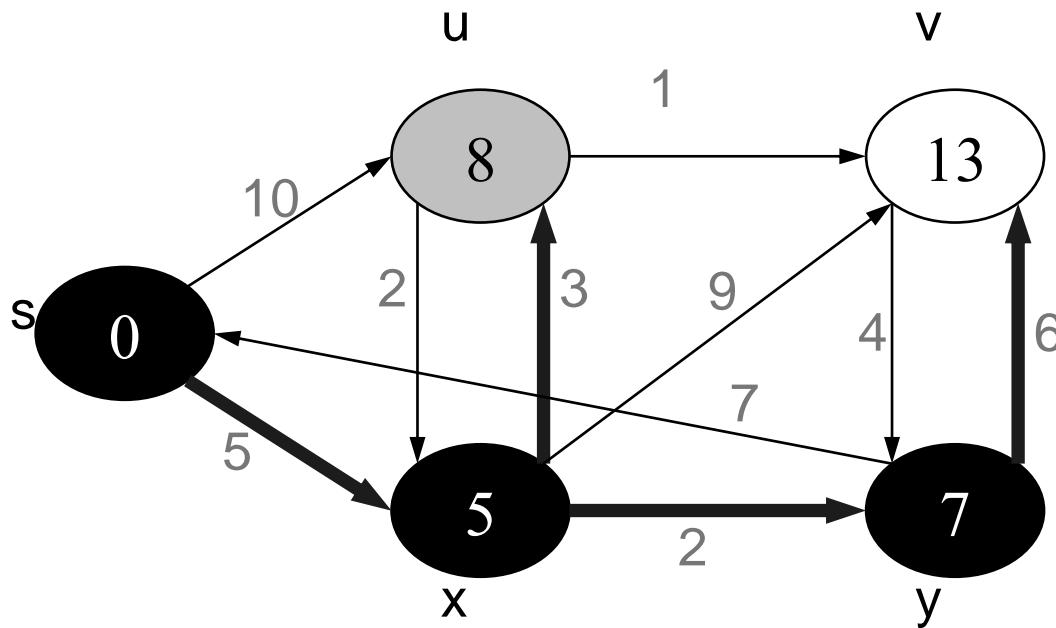
Dijkstra's Algorithm(Example)



$$S : \{ s, x \}$$

$$Q : \{ u, v, y \}$$

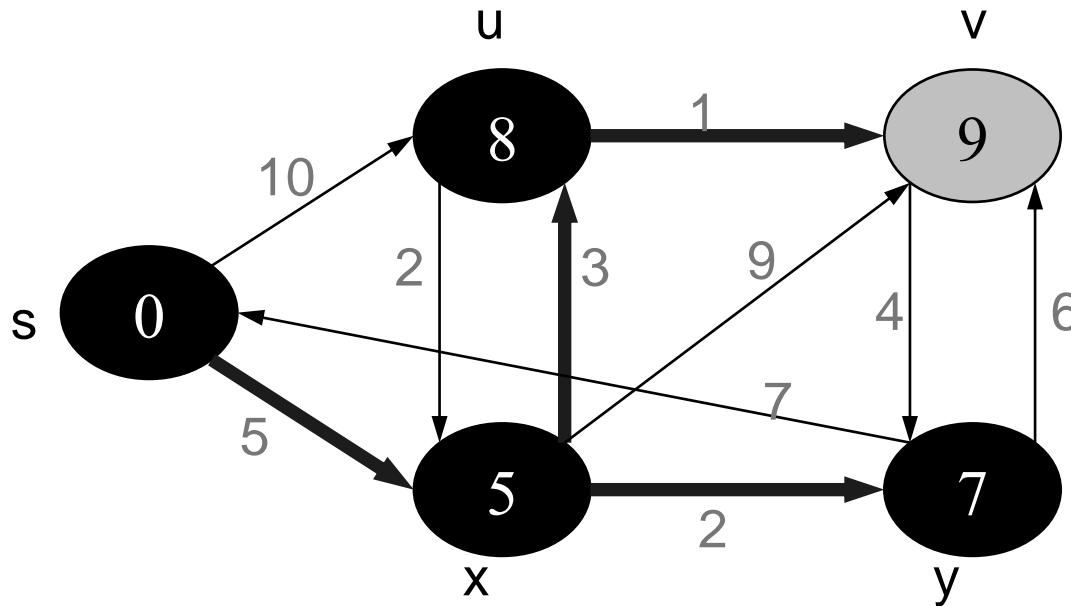
Dijkstra's Algorithm(Example)



$$S : \{ s, x, y \}$$

$$Q : \{ u, v \}$$

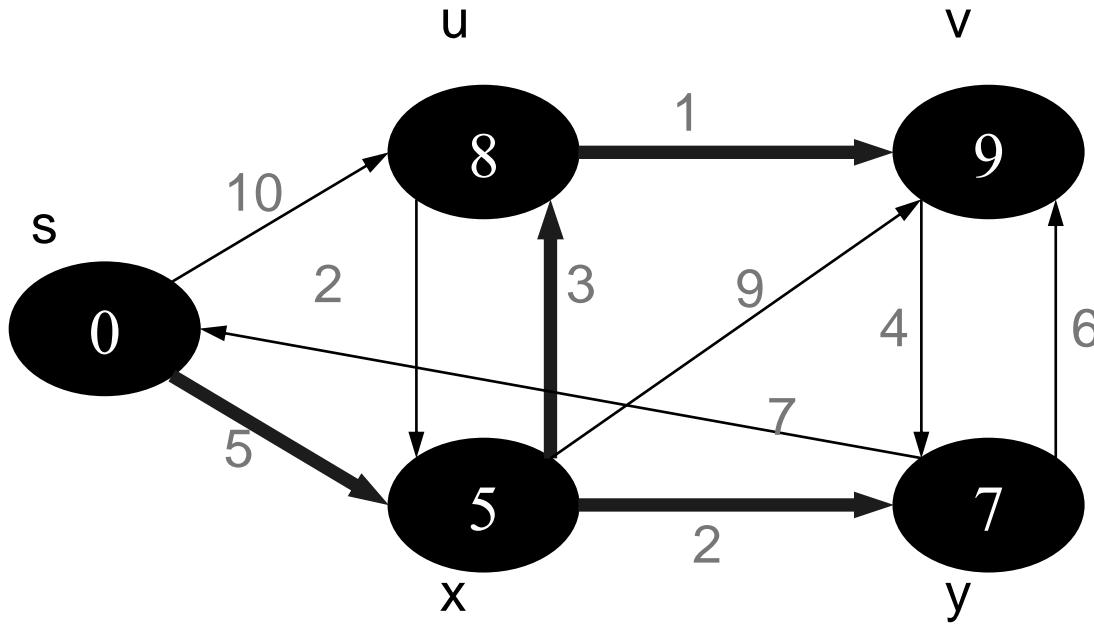
Dijkstra's Algorithm(Example)



$S : \{ s, x, y, u \}$

$Q : \{ v \}$

Dijkstra's Algorithm(Example)



$S : \{ s, x, y, u, v \}$

$Q : \{ \}$

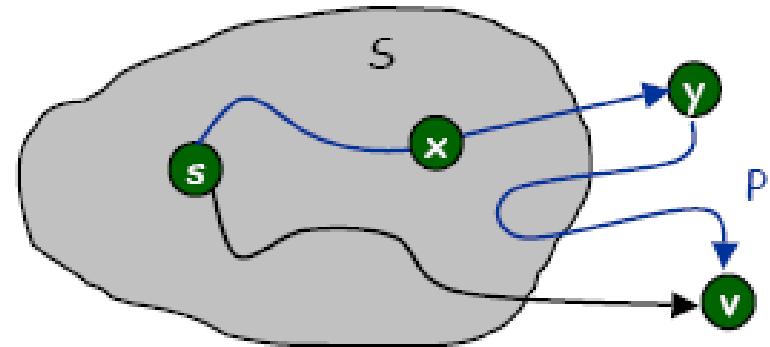
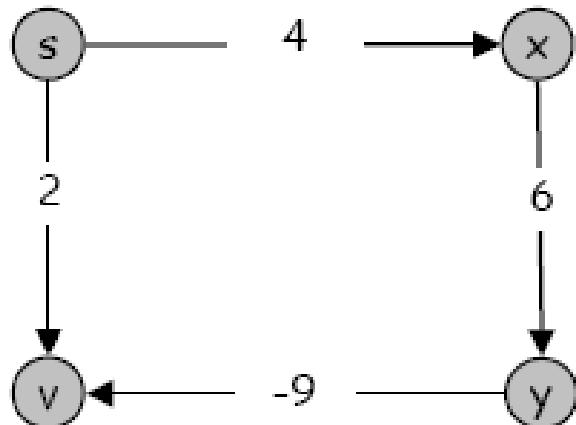
Now Q is empty.
So stop here

Dijkstra's Algorithm(Weakness)

Dijkstra's Algorithm With Negative Costs

Dijkstra's algorithm fails if there are negative weights.

- Ex: Selects vertex v immediately after s .
But shortest path from s to v is $s-x-y-v$.



Dijkstra proof of correctness breaks down since it assumes cost of P is nonnegative.

Dijkstra's Algorithm(Analysis)

- Maintains the min-priority queue Q with following operations
 - Insert
 - Extract-min
 - Decrease-key
- Running time depends on how the min-priority queue is implemented
 - With matrix and priority queue = $O(V^2)$
 - With Fibonacci Heap and adjacency list = $O(E + V \log V)$
 - With min-priority queue with binary min-heap = $O(E \log V)$

Bellman-ford's Algorithm

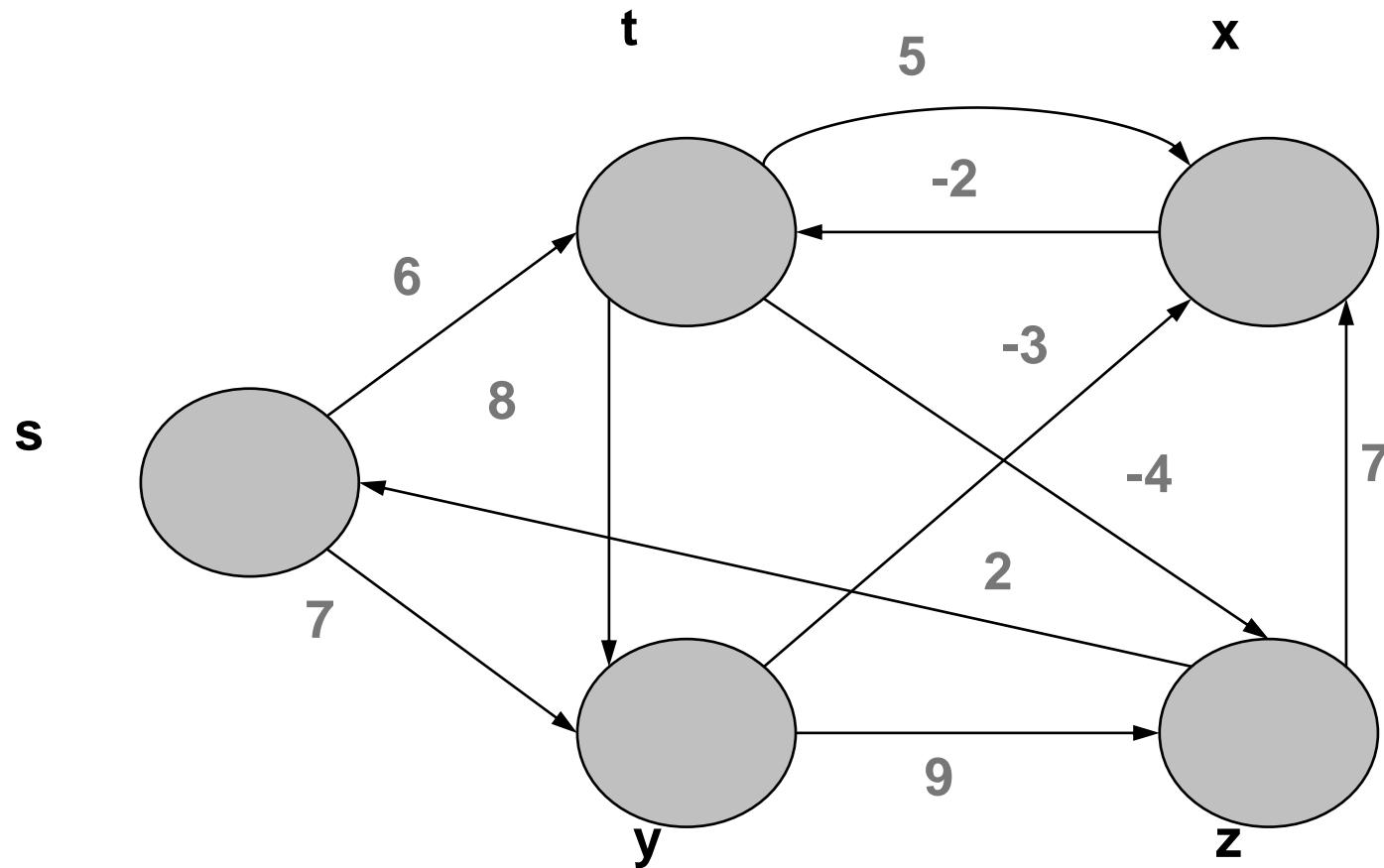
- Bellman-ford's algorithm allows negative edges
- This algorithm detects negative weight cycles reachable from source and returns *false(no solution)* otherwise returns the **shortest path-tree** (solution exists)
- Based on Dynamic Programming design approach.
- Basic Idea
 - Repeat the following $|V| - 1$ times
 - Relax each edge in E

Bellman-ford's Algorithm

Bellman-Ford (G, w, s)

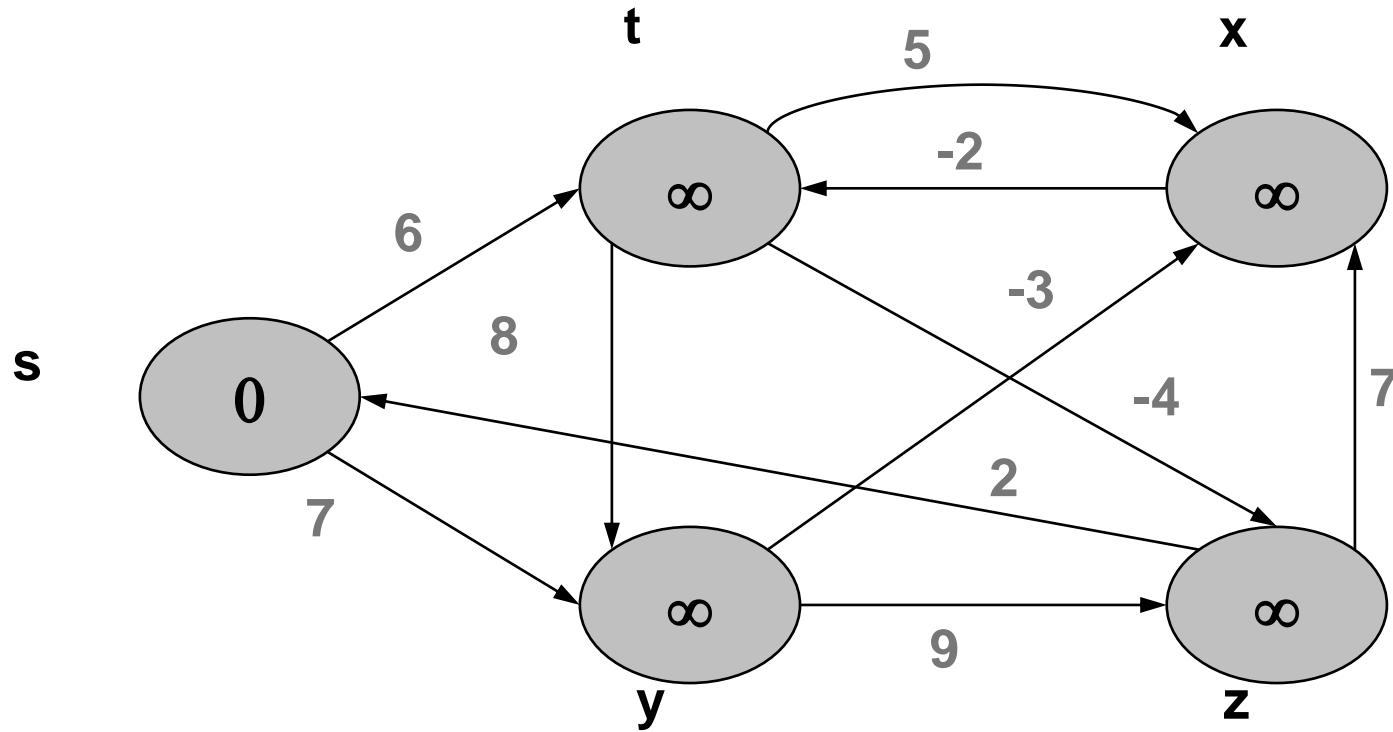
```
1 INITIALIZE-SINGLE-SOURCE ( $G, s$ )
2 for  $i \leftarrow 1$  to  $|V[G]| - 1$  do
3     for each edge  $(u, v) \in E[G]$  do
4         RELAX  $(u, v, w)$ 
5     for each edge  $(u, v) \in E[G]$  do
6         if  $d[v] > d[u] + w(u, v)$  then
7             return false
8 return true
```

Bellman-ford's Algorithm(Example)



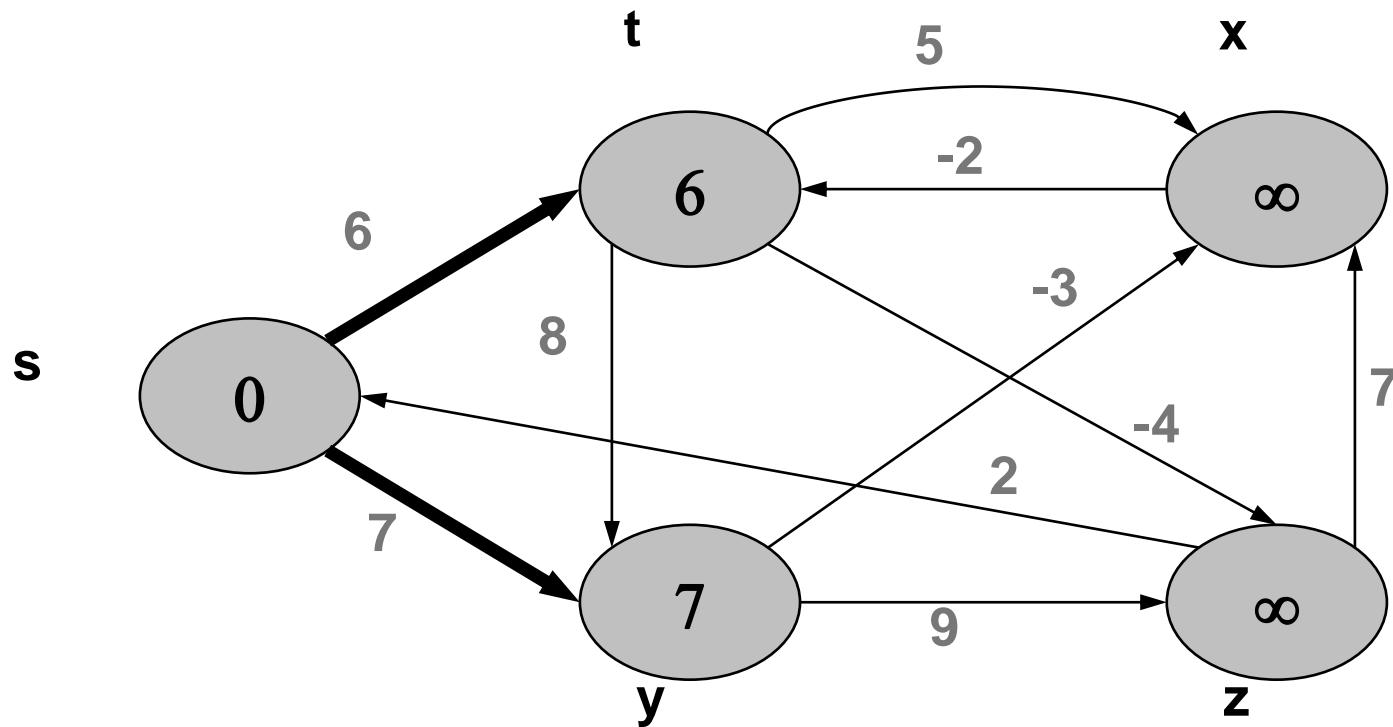
Bellman-ford's Algorithm(Example)

INITIALIZE-SINGLE-SOURCE (G, s)



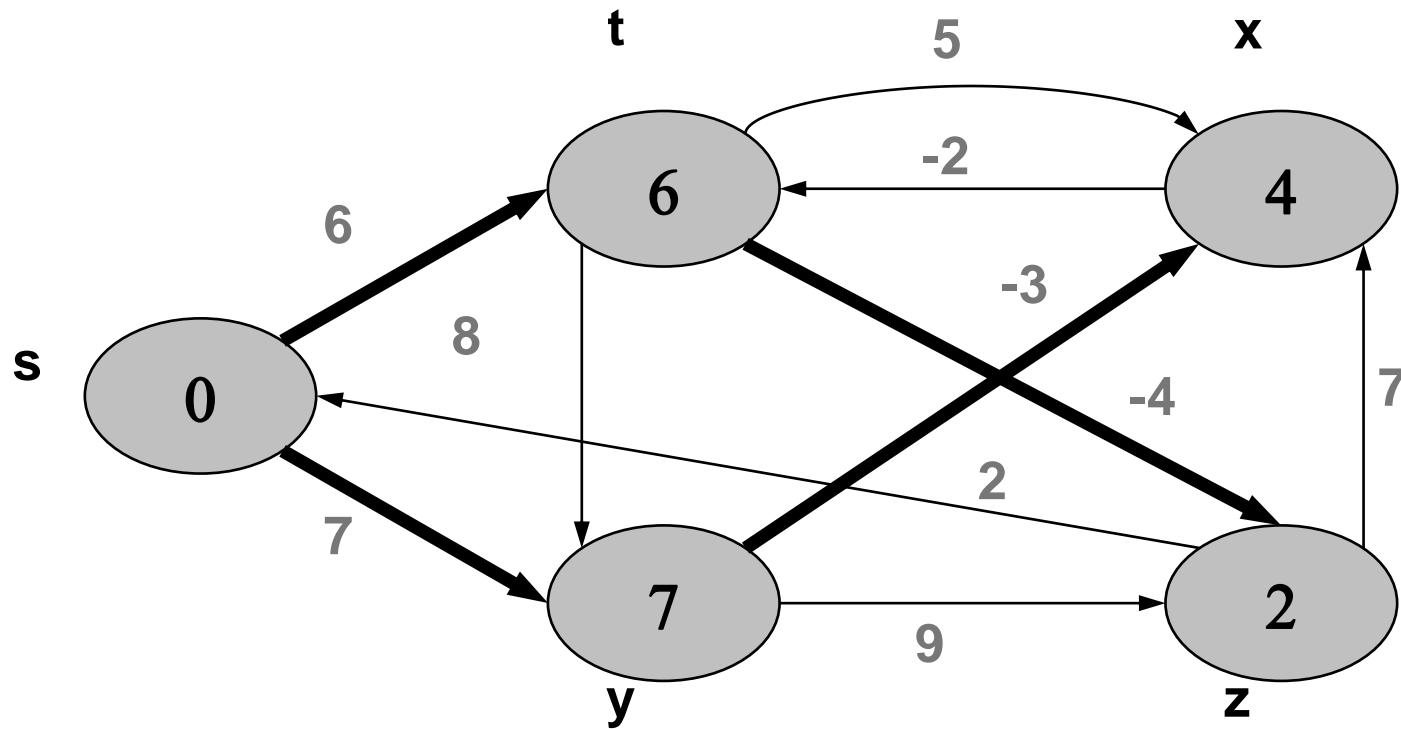
Bellman-ford's Algorithm(Example)

```
2 for i ← 1 to 4 do
3     for each edge (u, v) ∈ E[G] do
4         RELAX (u, v, w)
```



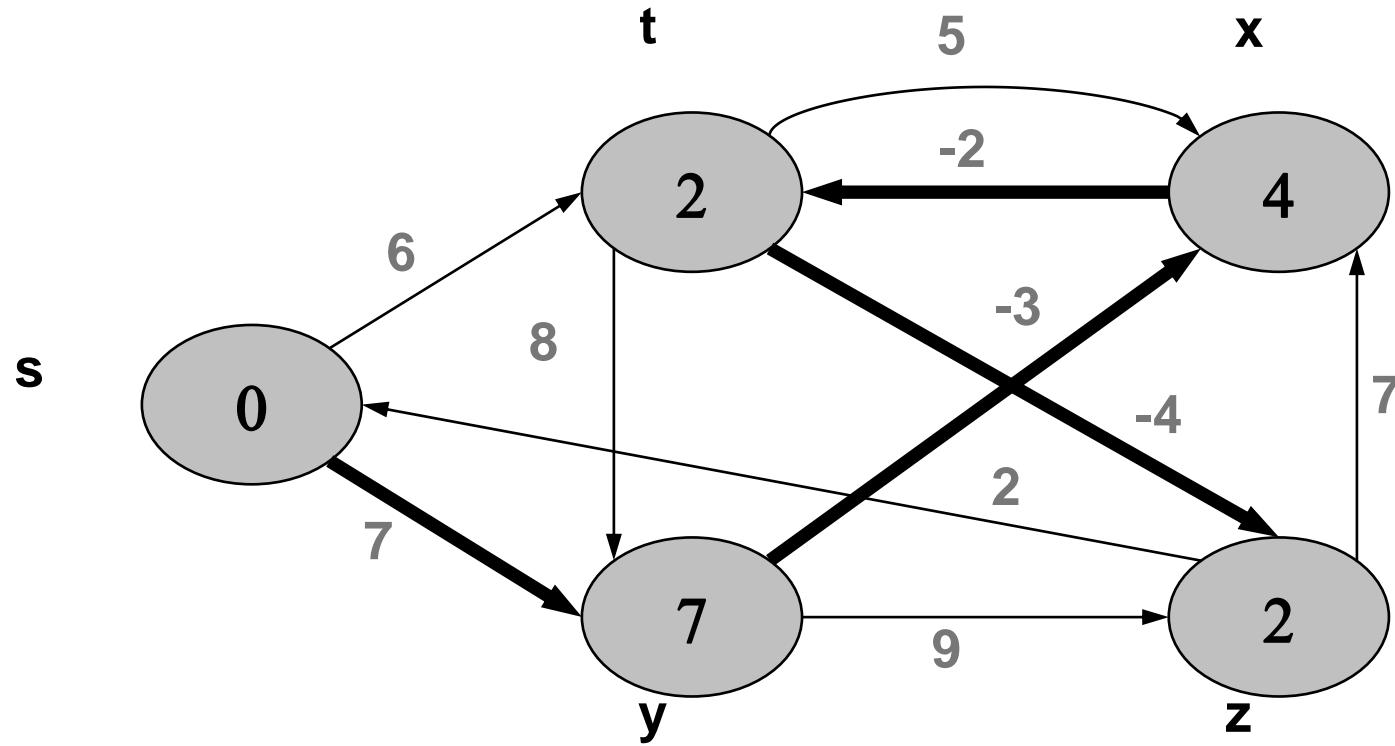
Bellman-ford's Algorithm(Example)

```
2 for i ← 2 to 4 do
3     for each edge (u, v) ∈ E[G] do
4         RELAX (u, v, w)
```



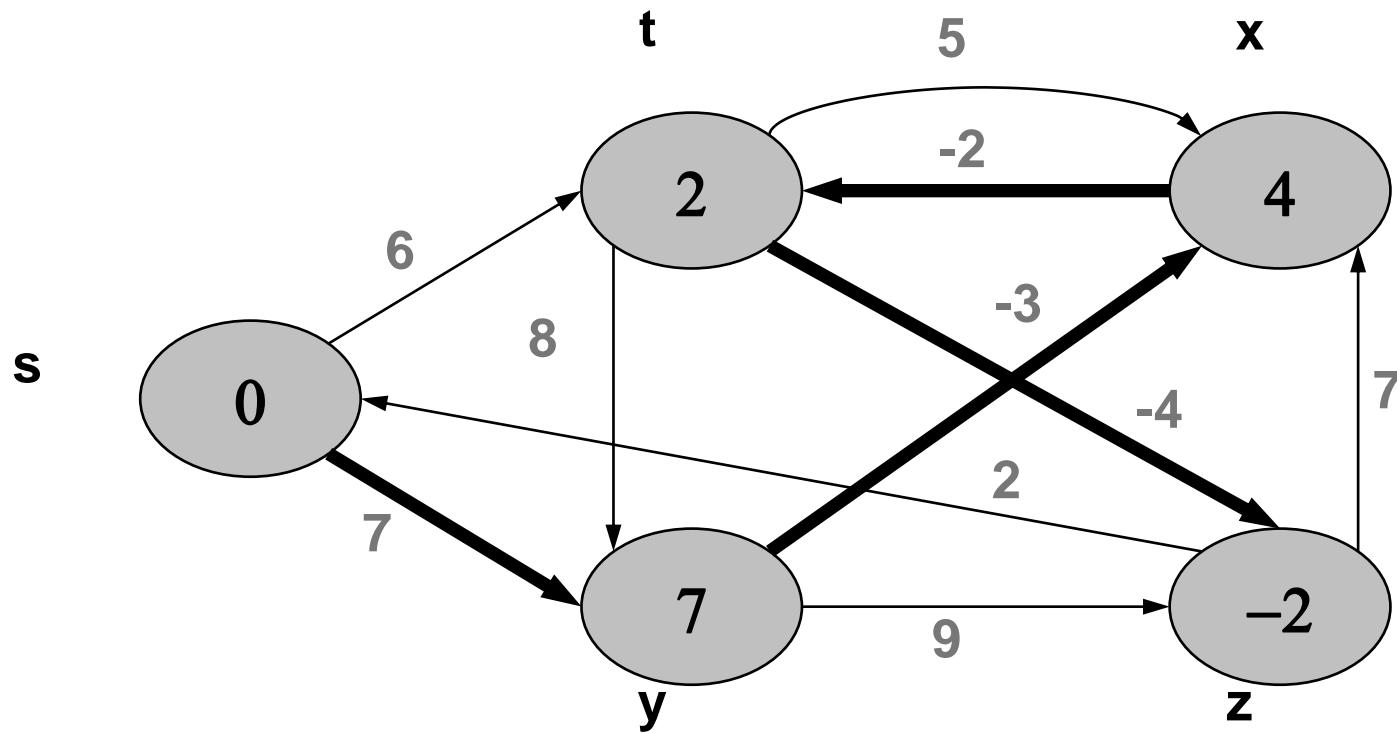
Bellman-ford's Algorithm(Example)

```
2 for i ← 3 to 4 do
3     for each edge (u, v) ∈ E[G] do
4         RELAX (u, v, w)
```



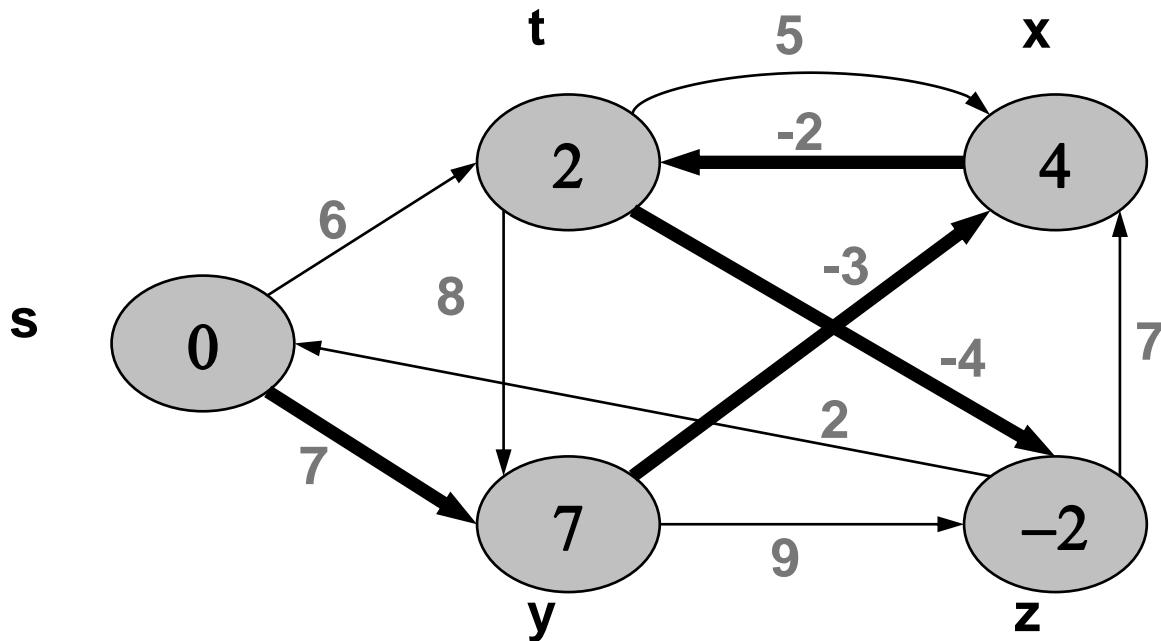
Bellman-ford's Algorithm(Example)

```
2 for i ← 4 to 4 do
3     for each edge (u, v) ∈ E[G] do
4         RELAX (u, v, w)
```



Bellman-ford's Algorithm(Example)

```
5 for each edge  $(u, v) \in E[G]$  do
6   if  $d[v] > d[u] + w(u, v)$  then
7     return false
```



There is no edge (u,v) belongs to edge set of the graph such that $d[v] > d[u] + w(u,v)$. Hence the algorithm returns true.

Bellman-ford's Algorithm(Running Time)

- Initialization = $\theta(V)$
- Each $|V| - 1$ passes over the edges in lines 2-4 takes $\theta(E)$ time
- for loop of lines 5-7 takes $O(E)$ times

=> So Running time = $O(VE)$

All-Pair Shortest Path

Floyd-Warshall's Algorithm

- Negative weight edges may be present, but we assume that there is no negative-weight cycles.
- We follow the dynamic-programming process

Considers the “**intermediate**” vertices of a shortest path, where an intermediate vertex of a simple path

$p = \langle v_1, v_2, v_3 \dots, v_l \rangle$ is any vertex other than v_1 and v_l
i.e. any vertex in the set $\{v_2, v_3 \dots, v_{l-1}\}$

Floyd-Warshall's Algorithm

Observation:

- The vertices of G are $V = \{1, 2, \dots, n\}$
- Let us consider a subset $\{1, 2, \dots, k\}$
- For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \dots, k\}$
- Let p be a minimum-weight path from among them.
- Floyd-Warshall algorithm exploit the relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \dots, k - 1\}$
- The relationship depends on whether k is an intermediate vertex of path p

Floyd-Warshall's Algorithm

Observation:

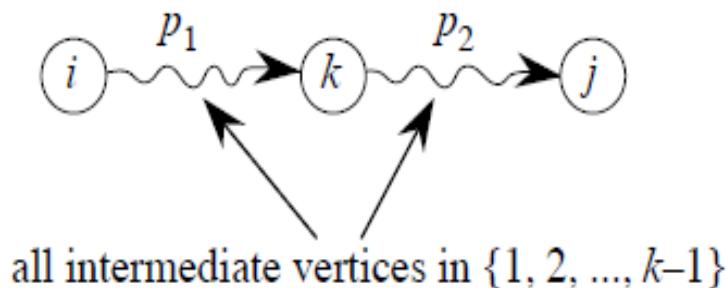
- If k is not an intermediate vertex of path p , then all intermediate vertices are in the set $\{1, 2, \dots, k - 1\}$. Thus a shortest path from i to j with intermediate vertices in the set $\{1, 2, \dots, k - 1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \dots, k\}$
- If k is an intermediate vertex of path p , then we break p down into

Floyd-Warshall's Algorithm

$d_{ij}^{(k)}$ = shortest path weight from vertex $i \rightsquigarrow j$ with all intermediate vertices $\{1, 2, \dots, k\}$

Consider a shortest path $i \xrightarrow{p} j$ with all intermediate vertices in $\{1, 2, \dots, k\}$:

- If **k is not an intermediate vertex**, then all intermediate vertices of p are in $\{1, 2, \dots, k-1\}$
- If **k is an intermediate vertex**:



Disjoint Sets Data Structure (Chap. 21)

- A disjoint-set is a collection $S = \{S_1, S_2, \dots, S_k\}$ of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*.
- Disjoint set operations:
 - MAKE-SET(x): create a new set with only x . assume x is not already in some other set.
 - UNION(x, y): combine the two sets containing x and y into one new set. A new representative is selected.
 - FIND-SET(x): return the representative of the set containing x .

Multiple Operations

- Suppose multiple operations:
 - n : #MAKE-SET operations (executed at beginning).
 - m : #MAKE-SET, UNION, FIND-SET operations.
 - $m \geq n$, #UNION operation is at most $n-1$.

An Application of Disjoint-Set

- Determine the connected components of an undirected graph.

CONNECTED-COMPONENTS(G)

1. **for** each vertex $v \in V[G]$
2. **do** MAKE-SET(v)
3. **for** each edge $(u,v) \in E[G]$
4. **do if** FIND-SET(u) \neq FIND-SET(v)
5. **then** UNION(u,v)

SAME-COMPONENT(u,v)

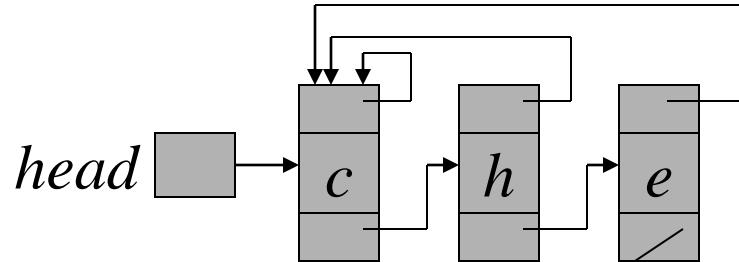
1. **if** FIND-SET(u) = FIND-SET(v)
2. **then return** TRUE
3. **else return** FALSE

Linked-List Implementation

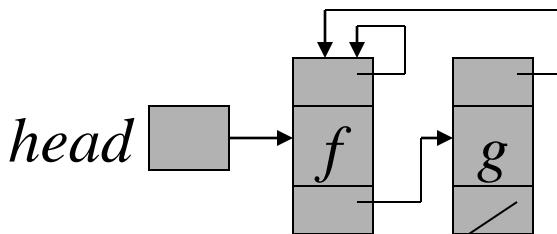
- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.
- Example:
- MAKE-SET costs $O(1)$: just create a single element list.
- FIND-SET costs $O(1)$: just return back-to-representative pointer.

Linked-lists for two sets

Set $\{c, h, e\}$

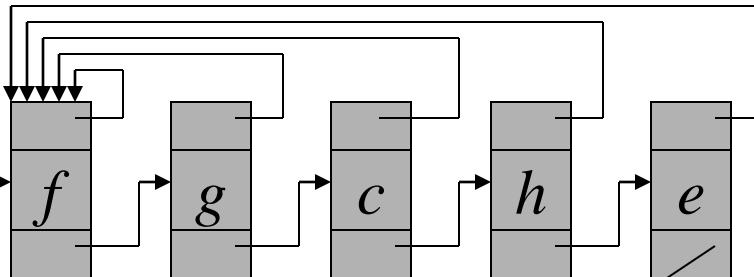


Set $\{f, g\}$



UNION of
two Sets

head



UNION Implementation

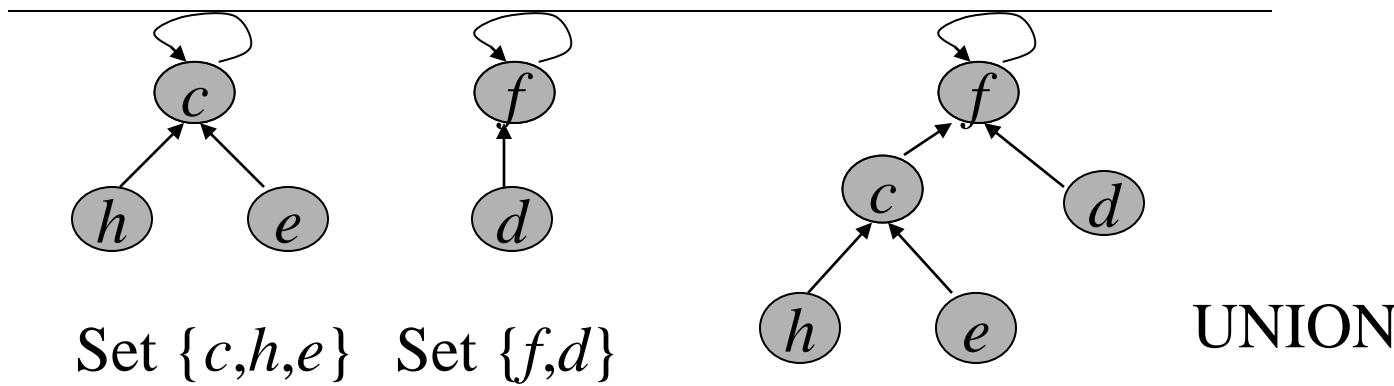
- A simple implementation: $\text{UNION}(x,y)$ just appends x to the end of y , updates all back-to-representative pointers in x to the head of y .
- Each UNION takes time linear in the x 's length.
- Suppose n $\text{MAKE-SET}(x_i)$ operations ($O(1)$ each) followed by $n-1$ UNION
 - $\text{UNION}(x_1, x_2)$, $O(1)$,
 - $\text{UNION}(x_2, x_3)$, $O(2)$,
 -
 - $\text{UNION}(x_{n-1}, x_n)$, $O(n-1)$
- The UNIONS cost $1+2+\dots+n-1=\Theta(n^2)$
- So $2n-1$ operations cost $\Theta(n^2)$, average $\Theta(n)$ each.
- Not good!! How to solve it ???

Weighted-Union Heuristic

- Instead appending x to y , appending the shorter list to the longer list.
- Associated a length with each list, which indicates how many elements in the list.
- Result: a sequence of m MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations, the running time is $O(m+n\lg n)$. Why???
- Hints: Count the number of updates to back-to-representative pointer for any x in a set of n elements. Consider that each time, the UNION will at least double the length of united set, it will take at most $\lg n$ UNIONS to unite n elements. So each x 's back-to-representative pointer can be updated at most $\lg n$ times.

Disjoint-set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.



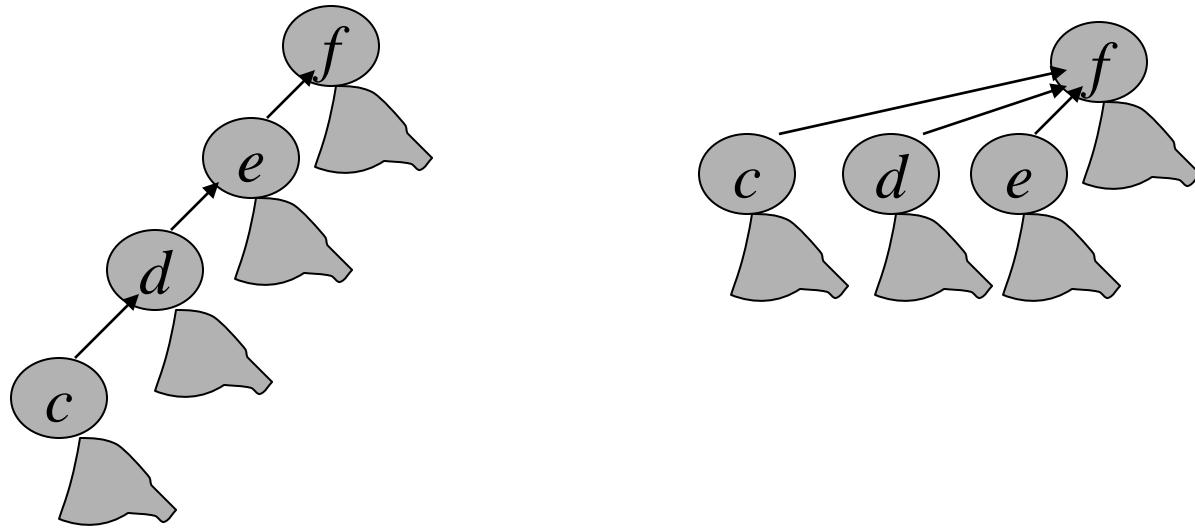
Straightforward Solution

- Three operations
 - $\text{MAKE-SET}(x)$: create a tree containing x . $O(1)$
 - $\text{FIND-SET}(x)$: follow the chain of parent pointers until to the root. $O(\text{height of } x\text{'s tree})$
 - $\text{UNION}(x,y)$: let the root of one tree point to the root of the other. $O(1)$
- It is possible that $n-1$ UNION s results in a tree of height $n-1$. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

Union by Rank & Path Compression

- Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.
- Path Compression: used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

Path Compression



Algorithm for Disjoint-Set Forest

MAKE-SET(x)

1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

UNION(x,y)

1. LINK(FIND-SET(x),FIND-SET(y))

LINK(x,y)

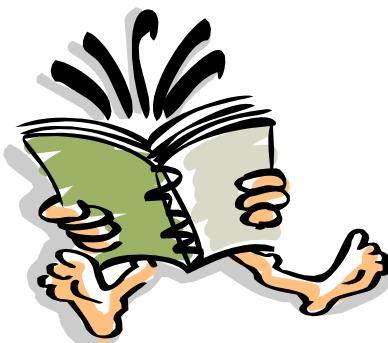
1. **if** $rank[x] > rank[y]$
2. **then** $p[y] \leftarrow x$
3. **else** $p[x] \leftarrow y$
4. **if** $rank[x] = rank[y]$
5. **then** $rank[y]++$

FIND-SET(x)

1. **if** $x \neq p[x]$
2. **then** $p[x] \leftarrow \text{FIND-SET}(p[x])$
3. **return** $p[x]$

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is:
 $O(m\alpha(n))$ where $\alpha(n) \leq 4$. So nearly linear in m .

Minimum Spanning Tree (MST)

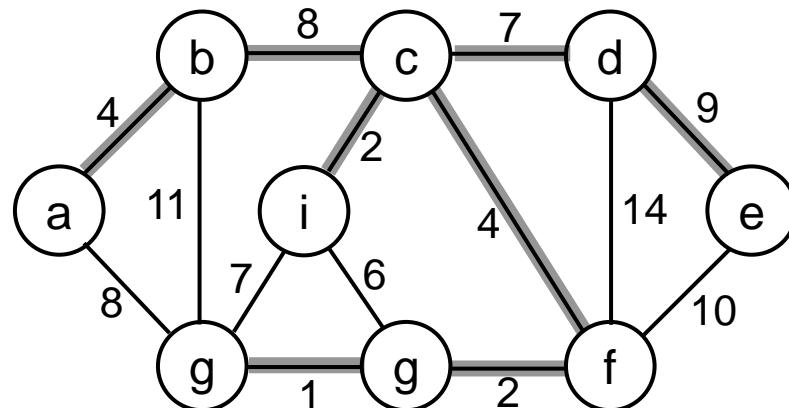


Ajit Kumar Behera

Minimum Spanning Trees

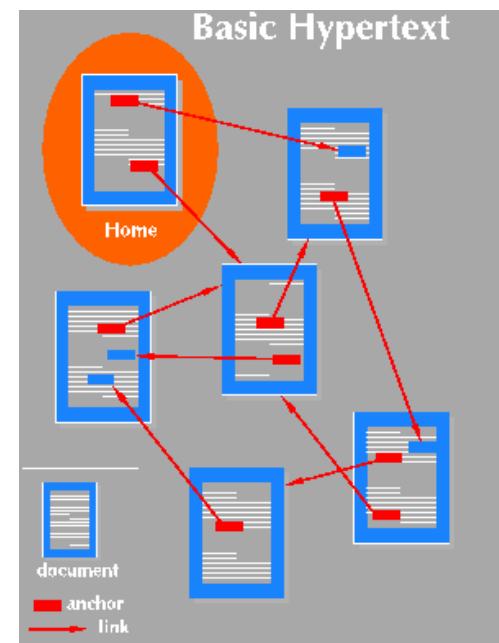
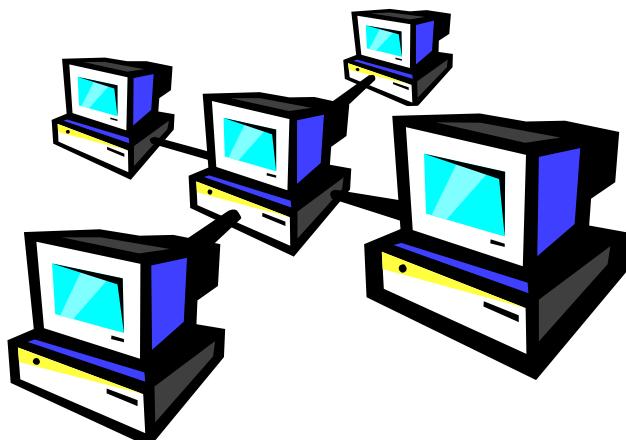
Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a connected undirected graph, where each edge $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$, and we have a weight $w(\mathbf{u}, \mathbf{v})$ on each edge $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$

- **Spanning Tree**
 - A **tree T** (i.e., connected, acyclic graph) which contains all the vertices of the graph
- **Minimum Spanning Tree**
 - Spanning tree with the **minimum sum of weights**



Applications of MST

- Find the least expensive way to connect a set of cities, terminals, computers, etc.

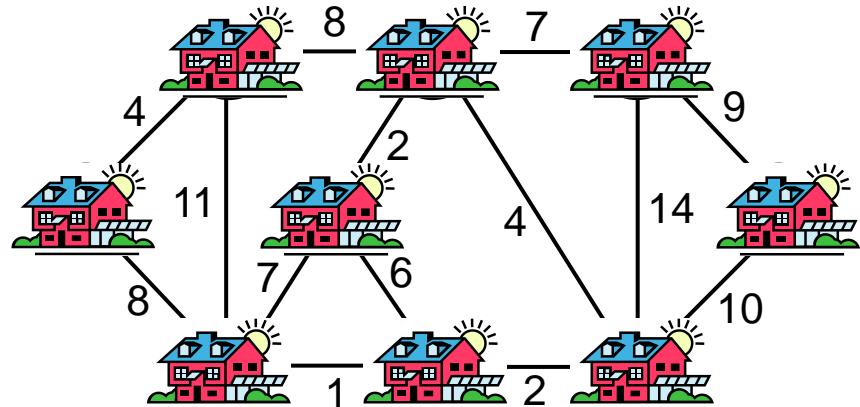


Minimum Spanning Trees

- A connected, undirected graph: $G = (V, E)$
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

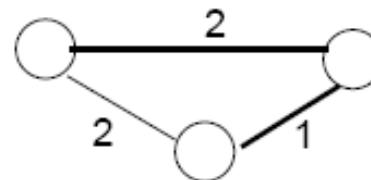
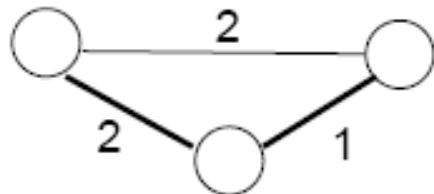
Find $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized



Properties of Minimum Spanning Trees

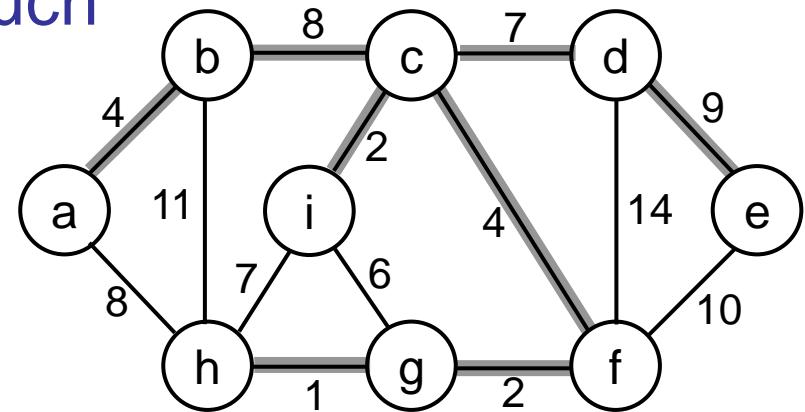
- Minimum spanning tree is **not** unique



- MST has no cycles :
 - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- No. of edges in a MST:
 - $|V| - 1$

Growing a MST – Generic Approach

- Grow a set A of edges (initially empty)
- Incrementally add edges to A such that they would belong to a MST



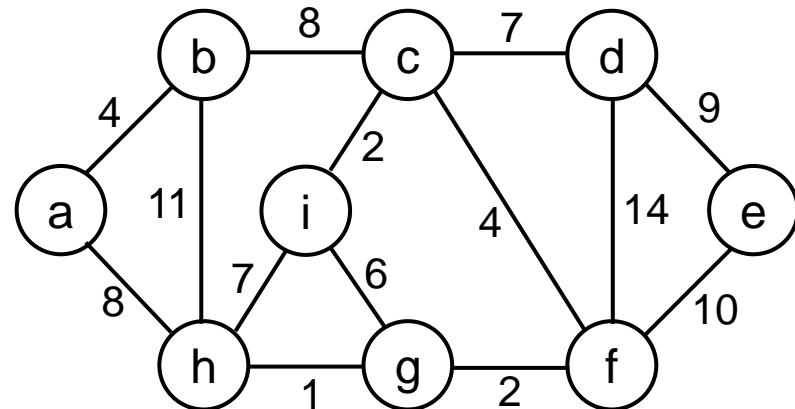
Idea: Add only “safe” edges

- An edge (u, v) is **safe** for A if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST

Generic MST algorithm

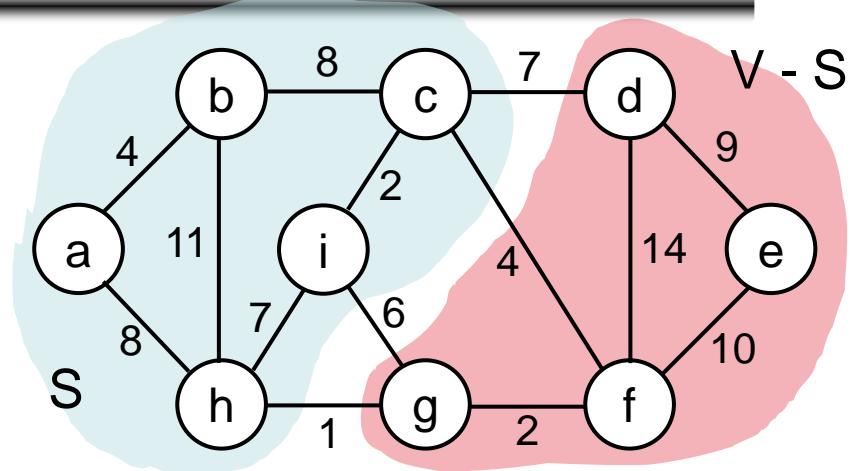
1. $A \leftarrow \emptyset$
2. **while** A does not form a spanning tree
3. **do** find an edge (u, v) that is **safe** for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A

- How do we find safe edges?



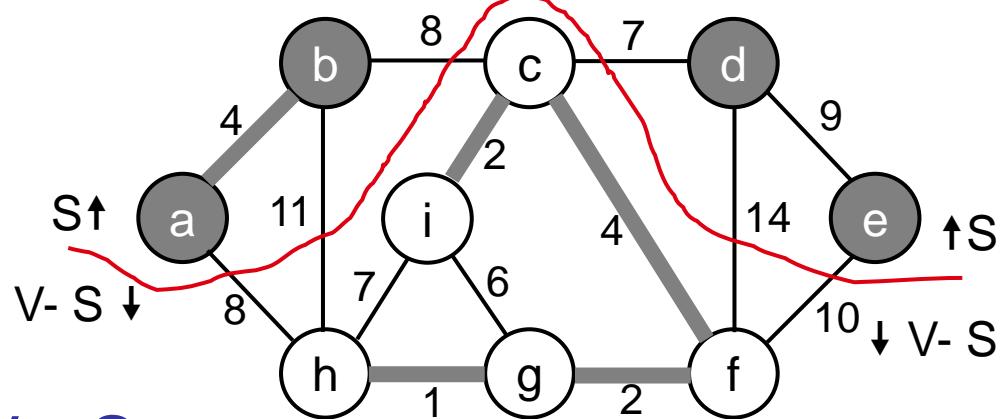
Finding Safe Edges

- Let's look at edge (h, g)
 - Is it safe for A initially?
- Later on:
 - Let $S \subset V$ be any set of vertices that includes h but not g (so that g is in $V - S$)
 - In any MST, there has to be one edge (at least) that connects S with $V - S$
 - Why not choose the edge with **minimum weight** (h,g) ?



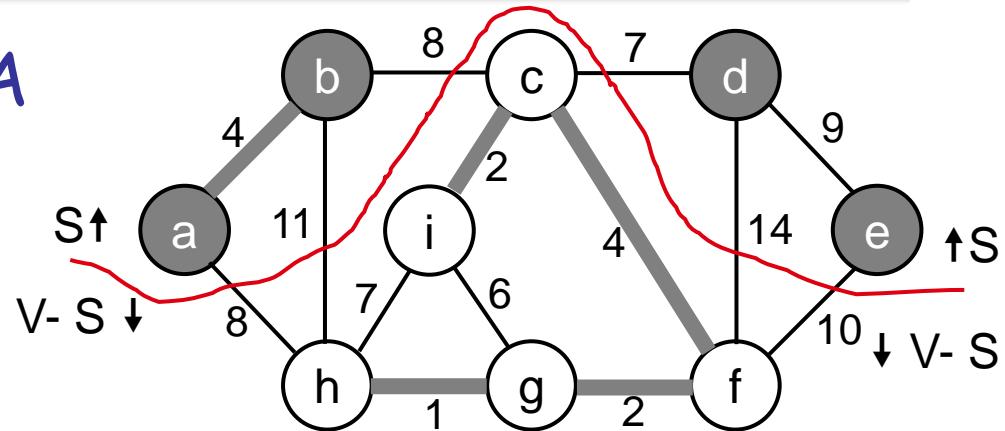
Definitions

- A **cut** $(S, V - S)$ is a partition of vertices into disjoint sets S and $V - S$
- An edge **crosses** the cut $(S, V - S)$ if one endpoint is in S and the other in $V - S$



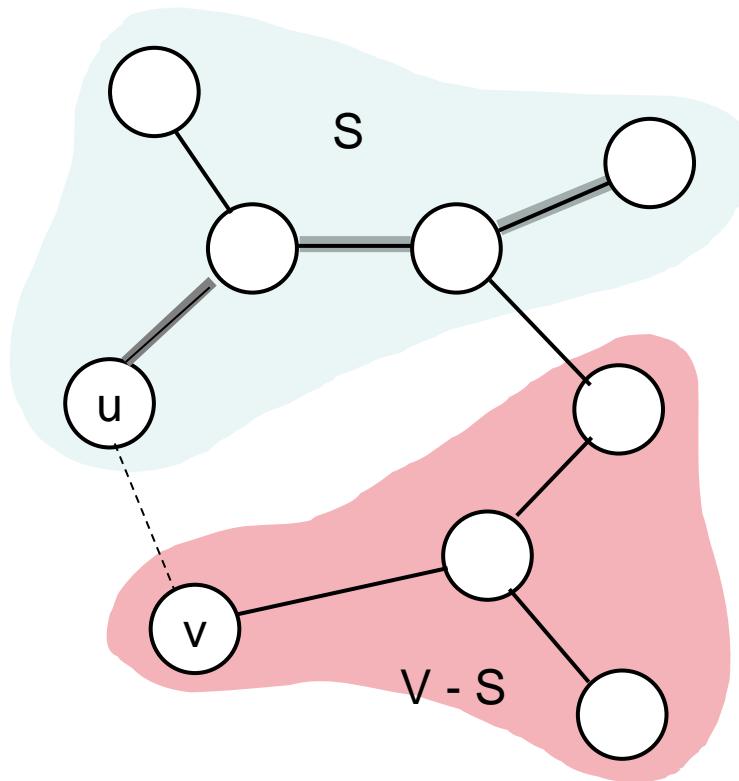
Definitions (cont'd)

- A cut **respects** a set A of edges \Leftrightarrow no edge in A crosses the cut
- An edge is a **light edge** crossing a cut \Leftrightarrow its weight is minimum over all edges crossing the cut
 - Note that for a given cut, there can be > 1 light edges crossing it



Theorem

- Let A be a subset of some MST (i.e., T), $(S, V - S)$ be a **cut** that respects A , and (u, v) be a **light edge** crossing $(S, V - S)$. Then (u, v) is safe for A .



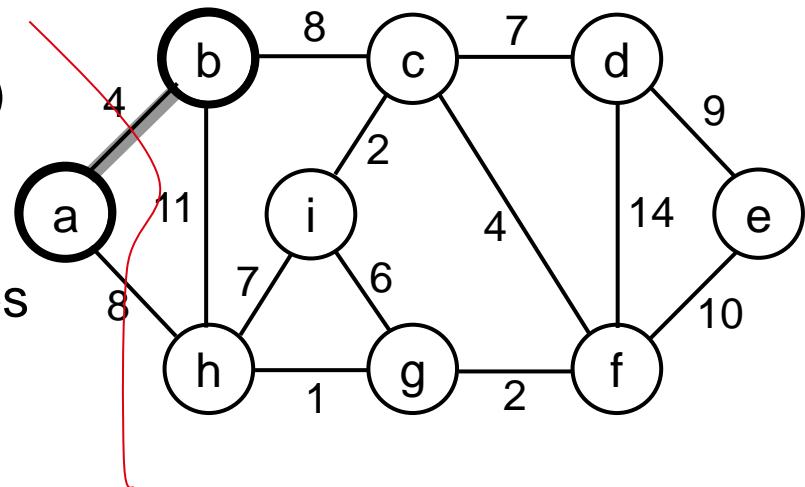
MST

Two greedy algorithms that implement generic approach:

- Kruskal's algorithm
 - Running time = $O(E \lg V)$
- Prim's algorithm
 - Running time = $O(E \lg V)$

Prim's Algorithm

- The edges in set A always form a single tree
- Starts from an arbitrary “root”: $V_A = \{a\}$
- At each step:
 - Find a light edge crossing $(V_A, V - V_A)$
 - Add this edge to A
 - Repeat until the tree spans all vertices

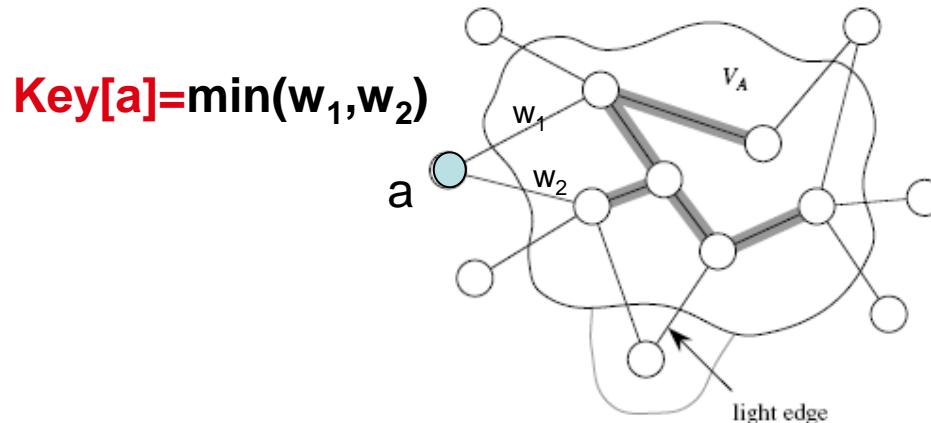
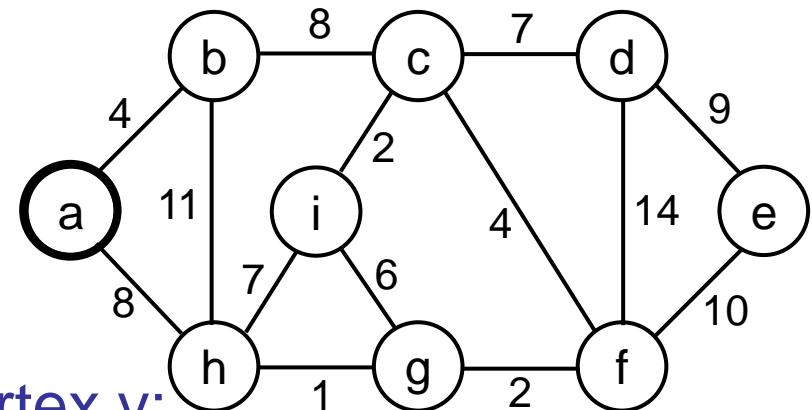


How to Find Light Edges Quickly?

Use a priority queue Q:

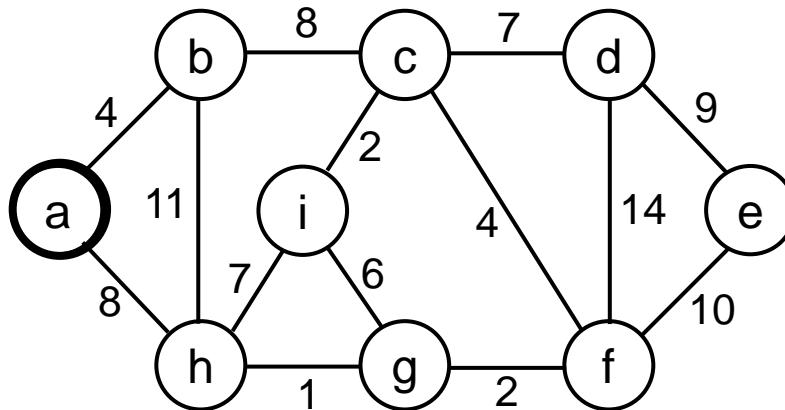
- Contains vertices not yet included in the tree, i.e., $(V - V_A)$
 - $V_A = \{a\}$, $Q = \{b, c, d, e, f, g, h, i\}$
- We associate a key with each vertex v:

$\text{key}[v] = \text{minimum weight of any edge } (u, v) \text{ where } u \in V_A$



How to Find Light Edges Quickly? (cont.)

- After adding a new node to V_A we update the weights of all the nodes adjacent to it
e.g., after adding a to the tree, $\text{key}[b]=4$ and $\text{key}[h]=8$
- Key of v is ∞ if v is not adjacent to any vertices in V_A

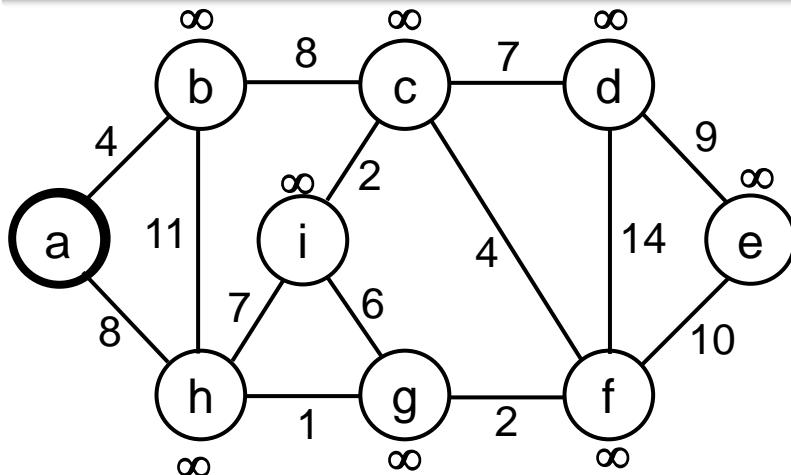


How to Find Light Edges Quickly? (cont.)

The edge of A will form a rooted tree with root r

- r is given as an input to the algorithm, but it can be any vertex
- Each vertex knows its parent by $\pi[v] = \text{parent of } v$
 $\pi[b] = \text{NIL}$, if $v = r$ or r has no parent
- As algorithm progresses, $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$
- At termination, $V_A = V \Rightarrow Q = \emptyset$,
so MST is $A = \{(v, \pi[v]) : v \in V - \{r\}\}$

Example

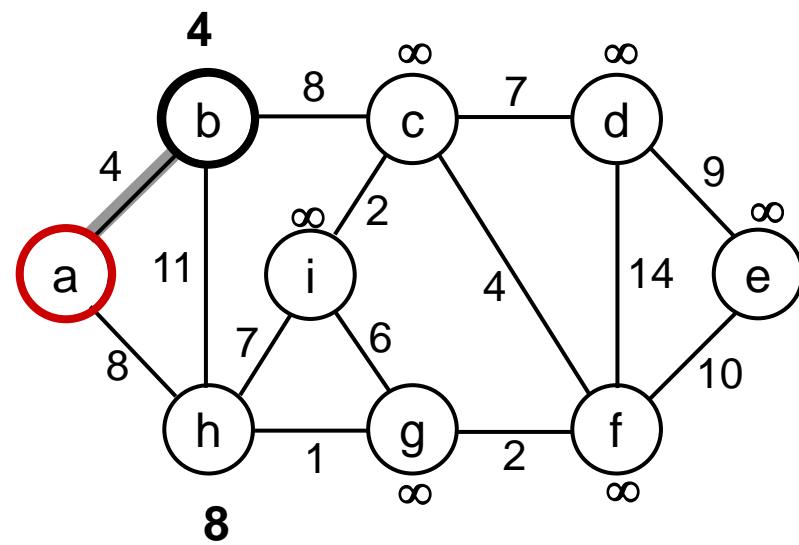


0 $\infty \infty \infty \infty \infty \infty \infty \infty$

$Q = \{a, b, c, d, e, f, g, h, i\}$

$V_A = \emptyset$

$\text{Extract-MIN}(Q) \Rightarrow a$



key [b] = 4 π [b] = a

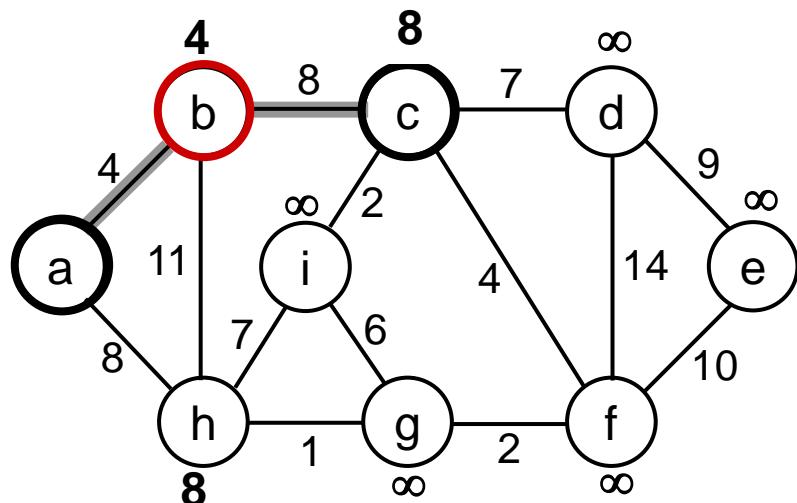
key [h] = 8 π [h] = a

4 $\infty \infty \infty \infty \infty 8 \infty$

$Q = \{b, c, d, e, f, g, h, i\}$ $V_A = \{a\}$

$\text{Extract-MIN}(Q) \Rightarrow b$

Example



key [c] = 8 π [c] = b
 key [h] = 8 π [h] = a - unchanged

8 ∞ ∞ ∞ 8 ∞

$Q = \{c, d, e, f, g, h, i\}$ $V_A = \{a, b\}$
 Extract-MIN(Q) $\Rightarrow c$

key [d] = 7 π [d] = c

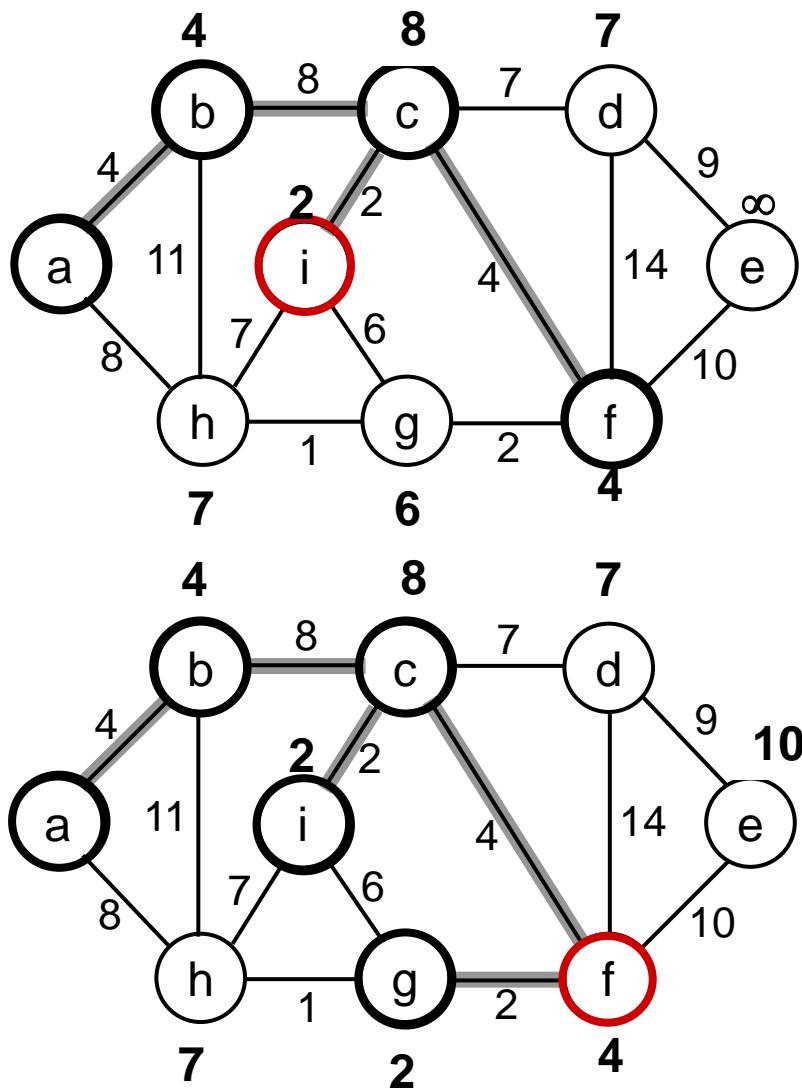
key [f] = 4 π [f] = c

key [i] = 2 π [i] = c

7 ∞ 4 ∞ 8 2

$Q = \{d, e, f, g, h, i\}$ $V_A = \{a, b, c\}$
 Extract-MIN(Q) $\Rightarrow i$

Example



key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

$Q = \{d, e, f, g, h\}$ $V_A = \{a, b, c, i\}$

Extract-MIN(Q) \Rightarrow f

key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

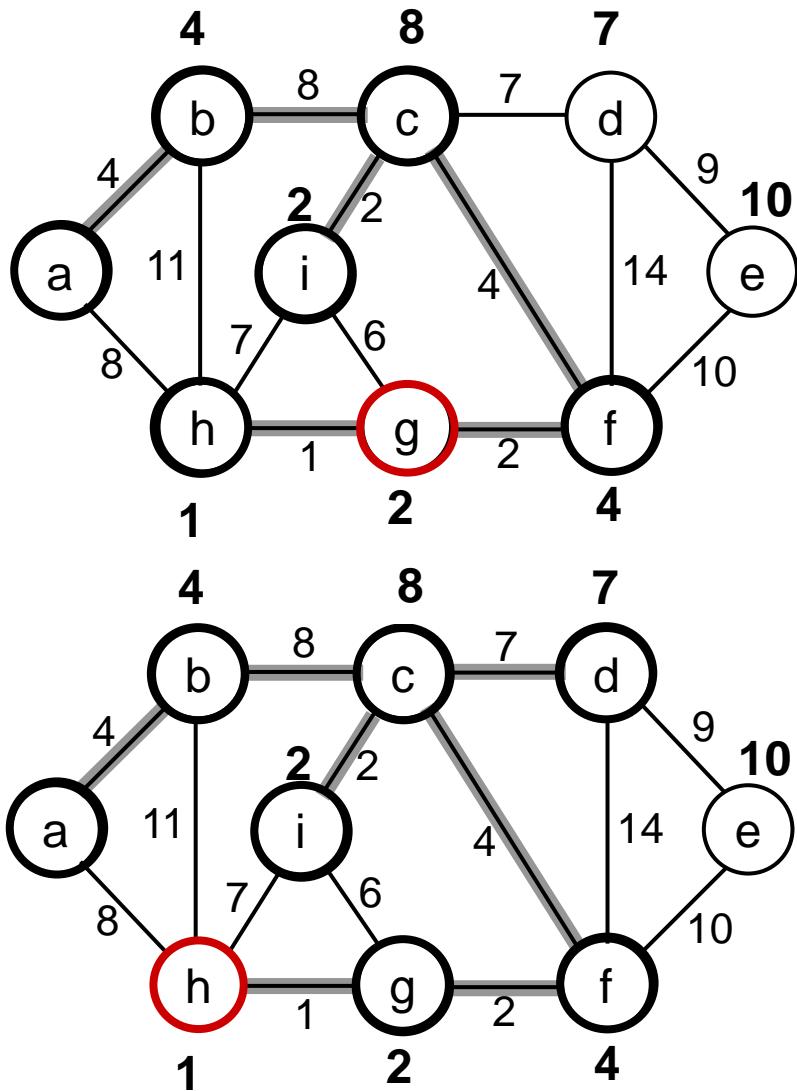
key [e] = 10 π [e] = f

7 10 2 8

$Q = \{d, e, g, h\}$ $V_A = \{a, b, c, i, f\}$

Extract-MIN(Q) \Rightarrow g

Example



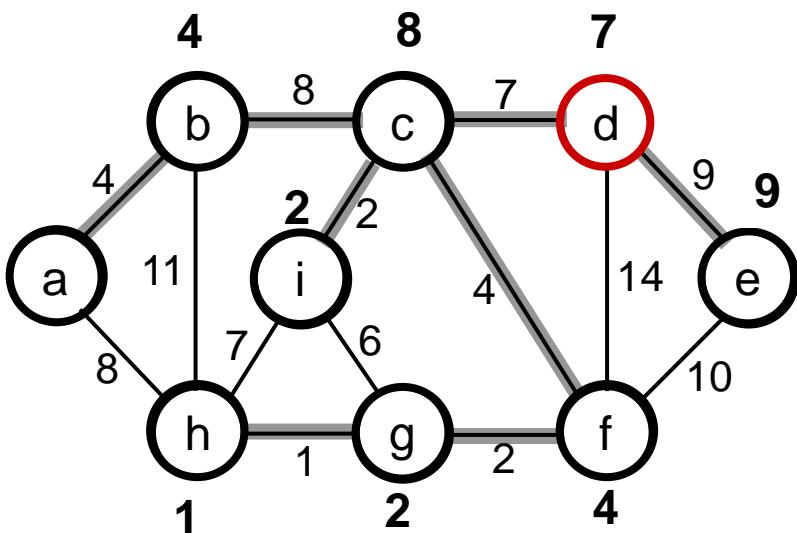
key [h] = 1 π [h] = g
7 10 1

$Q = \{d, e, h\}$ $V_A = \{a, b, c, i, f, g\}$
 $\text{Extract-MIN}(Q) \Rightarrow h$

7 10

$Q = \{d, e\}$ $V_A = \{a, b, c, i, f, g, h\}$
 $\text{Extract-MIN}(Q) \Rightarrow d$

Example



key [e] = 9 π [e] = f
9

$Q = \{e\}$ $V_A = \{a, b, c, i, f, g, h, d\}$

Extract-MIN(Q) $\Rightarrow e$

$Q = \emptyset$ $V_A = \{a, b, c, i, f, g, h, d, e\}$

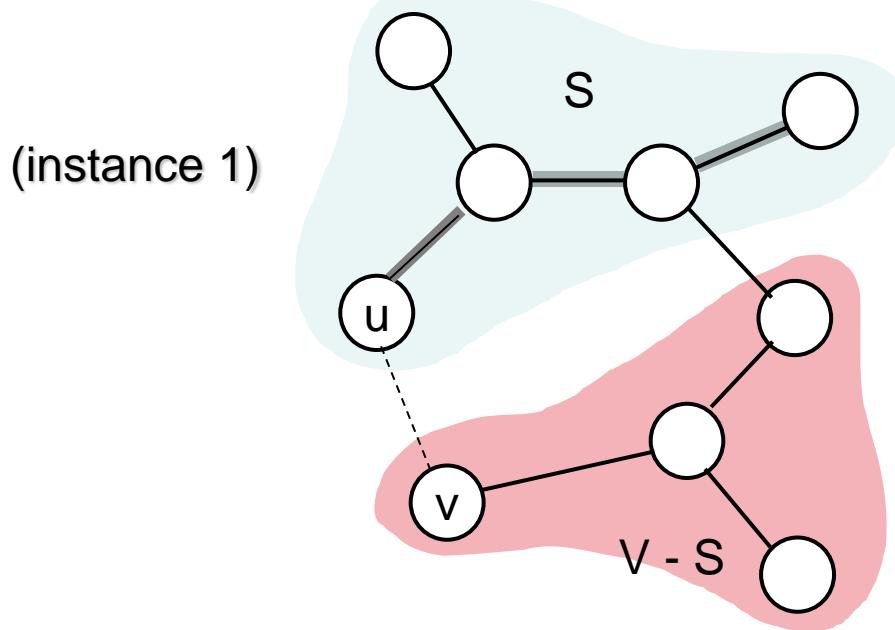
PRIM(V, E, w, r)

1. $Q \leftarrow \emptyset$
 2. **for** each $u \in V$
 3. **do** $\text{key}[u] \leftarrow \infty$
 4. $\pi[u] \leftarrow \text{NIL}$
 5. $\text{INSERT}(Q, u)$
 6. $\text{DECREASE-KEY}(Q, r, 0)$ ► $\text{key}[r] \leftarrow 0$ $\longleftarrow O(\lg V)$
 7. **while** $Q \neq \emptyset$ \longleftarrow Executed $|V|$ times
 8. **do** $u \leftarrow \text{EXTRACT-MIN}(Q)$ \longleftarrow Takes $O(\lg V)$
 9. **for** each $v \in \text{Adj}[u]$ \longleftarrow Executed $O(E)$ times total
 10. **do if** $v \in Q$ and $w(u, v) < \text{key}[v]$ \longleftarrow Constant
 11. **then** $\pi[v] \leftarrow u$ \longleftarrow Takes $O(\lg V)$
 12. $\text{DECREASE-KEY}(Q, v, w(u, v))$
- Total time:** $O(V\lg V + E\lg V) = O(E\lg V)$
- $O(V)$ if Q is implemented as a min-heap
- Min-heap operations: $O(V\lg V)$
- $O(E\lg V)$

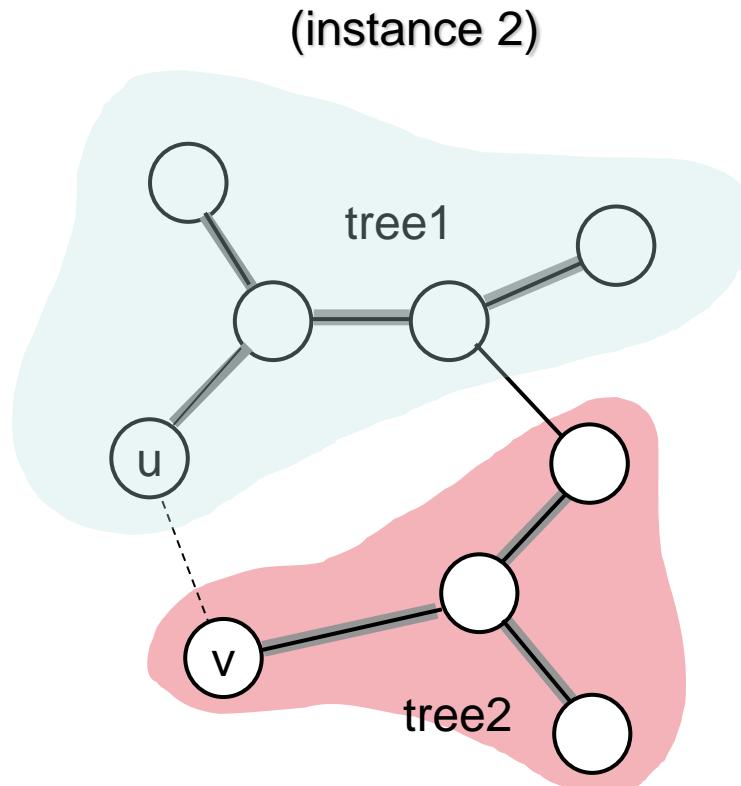
Prim's Algorithm

- Prim's algorithm is a “**greedy**” algorithm
 - Greedy algorithms find solutions based on a sequence of choices which are “**locally**” optimal at each step.
- Nevertheless, Prim's greedy strategy produces a globally optimum solution!
 - See proof for generic approach (i.e., slides 12-15)

A different instance of the generic approach



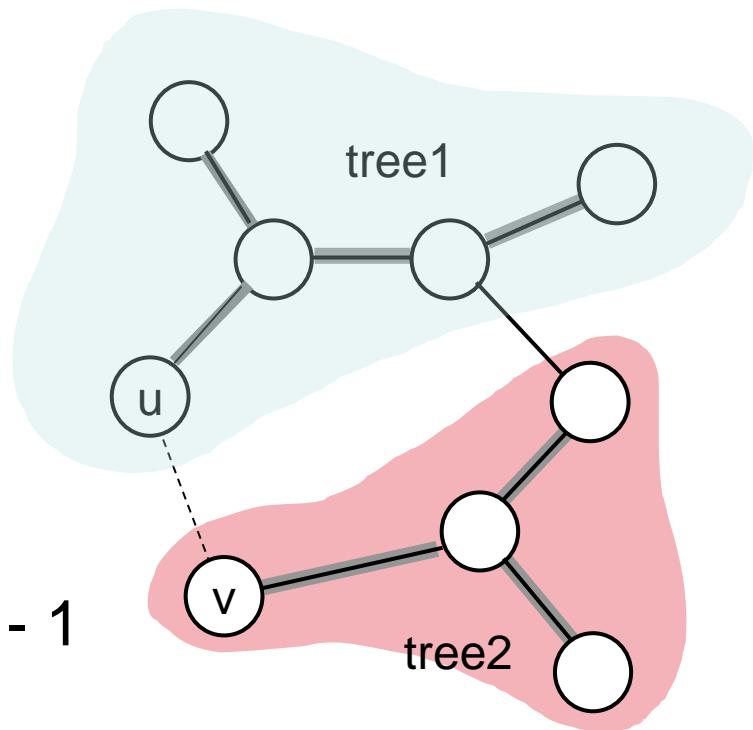
- **A is a forest containing connected components**
 - Initially, each component is a single vertex
- **Any safe edge merges two of these components into one**
 - Each component is a tree



Kruskal's Algorithm

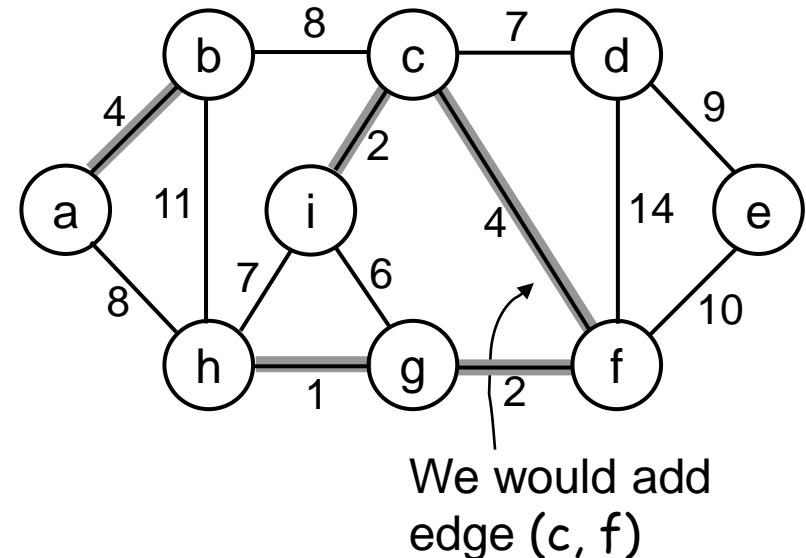
- How is it different from Prim's algorithm?

- Prim's algorithm grows one tree all the time
- Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
- Trees are merged together using **safe** edges
- Since an MST has exactly $|V| - 1$ edges, after $|V| - 1$ merges, we would have only one component

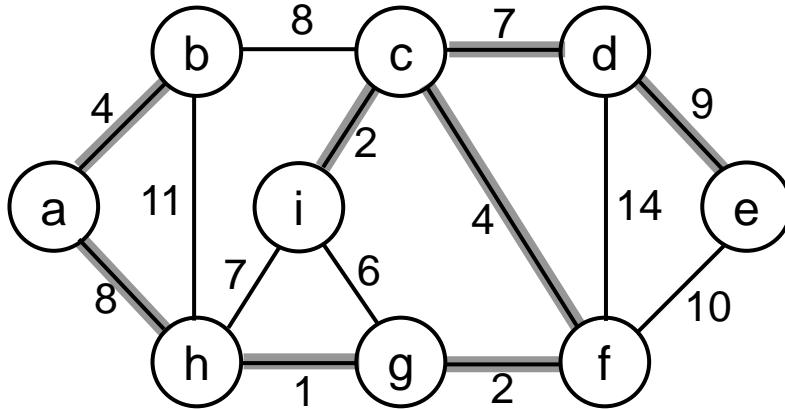


Kruskal's Algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them
- Which components to consider at each iteration?
 - Scan the set of edges in monotonically increasing order by weight



Example



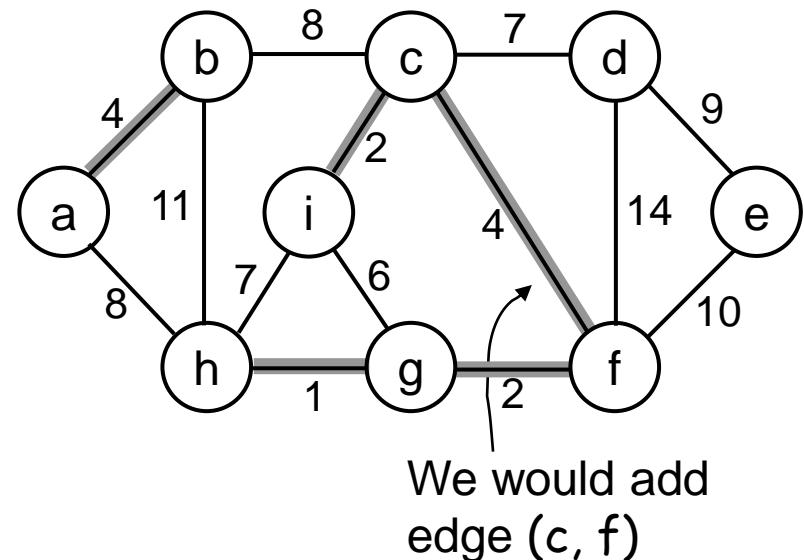
- 1: (h, g) 8: (a, h), (b, c)
 2: (c, i), (g, f) 9: (d, e)
 4: (a, b), (c, f) 10: (e, f)
 6: (i, g) 11: (b, h)
 7: (c, d), (i, h) 14: (d, f)

 {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1. Add (h, g) {g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2. Add (c, i) {g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3. Add (g, f) {g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4. Add (a, b) {g, h, f}, {c, i}, {a, b}, {d}, {e}
5. Add (c, f) {g, h, f, c, i}, {a, b}, {d}, {e}
6. Ignore (i, g) {g, h, f, c, i}, {a, b}, {d}, {e}
7. Add (c, d) {g, h, f, c, i, d}, {a, b}, {e}
8. Ignore (i, h) {g, h, f, c, i, d}, {a, b}, {e}
9. Add (a, h) {g, h, f, c, i, d, a, b}, {e}
10. Ignore (b, c) {g, h, f, c, i, d, a, b}, {e}
11. Add (d, e) {g, h, f, c, i, d, a, b, e}
12. Ignore (e, f) {g, h, f, c, i, d, a, b, e}
13. Ignore (b, h) {g, h, f, c, i, d, a, b, e}
14. Ignore (d, f) {g, h, f, c, i, d, a, b, e}

Implementation of Kruskal's Algorithm

- Uses a **disjoint-set** data structure to determine whether an edge connects vertices in different components



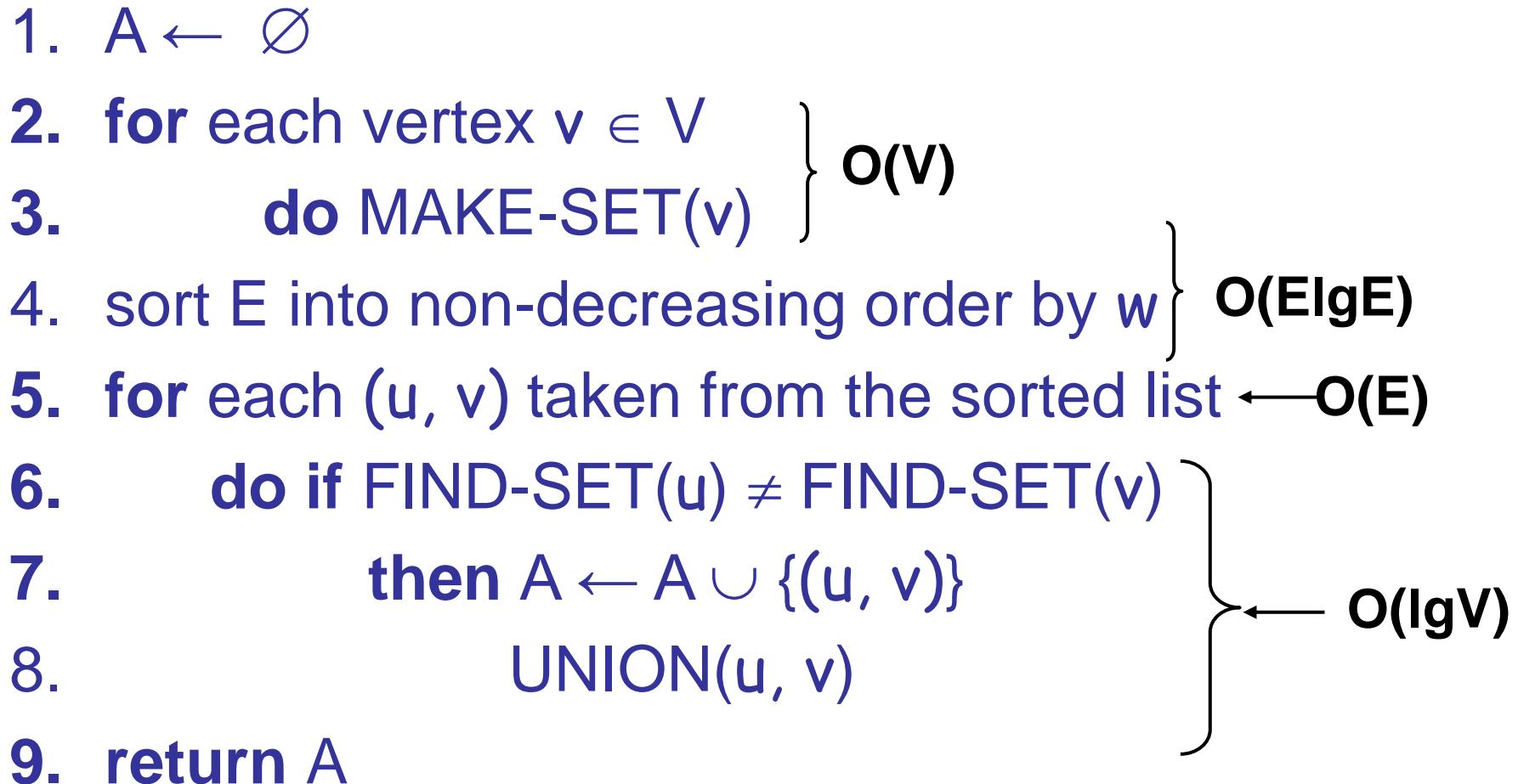
Operations on Disjoint Data Sets

- **MAKE-SET(u)** – creates a new set whose only member is u
- **FIND-SET(u)** – returns a representative element from the set that contains u
 - Any of the elements of the set that has a particular property
 - *E.g.:* $S_u = \{r, s, t, u\}$, the property is that the element be the first one alphabetically
 - $\text{FIND-SET}(u) = r$ $\text{FIND-SET}(s) = r$
 - FIND-SET has to return the same value for a given set

Operations on Disjoint Data Sets

- UNION(u, v) – unites the dynamic sets that contain u and v , say S_u and S_v
 - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$
$$\text{UNION } (u, v) = \{r, s, t, u, v, x, y\}$$
- Running time for FIND-SET and UNION depends on implementation.
- Can be shown to be $\alpha(n)=O(\lg n)$ where $\alpha()$ is a very slowly growing function (see **Chapter 21**)

KRUSKAL(V, E, w)

1. $A \leftarrow \emptyset$
 2. **for** each vertex $v \in V$
 3. **do** $\text{MAKE-SET}(v)$
 4. sort E into non-decreasing order by w
 5. **for** each (u, v) taken from the sorted list $\leftarrow O(E)$
 6. **do if** $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
 7. **then** $A \leftarrow A \cup \{(u, v)\}$
 8. $\text{UNION}(u, v)$
 9. **return** A
- 

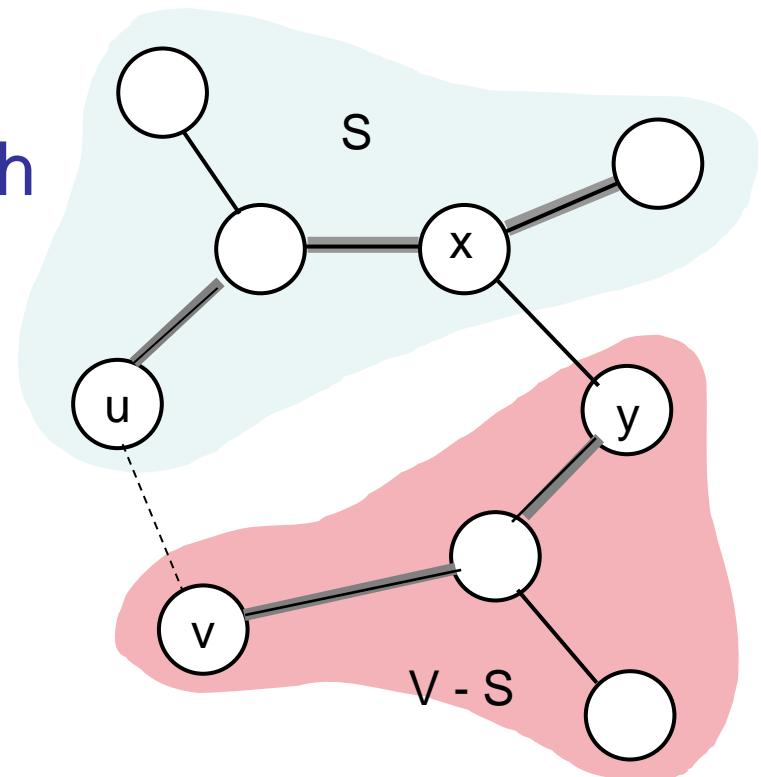
Running time: $O(V + E \lg E + E \lg V) = O(E \lg E)$ – dependent on the implementation of the disjoint-set data structure

KRUSKAL(V, E, w) (cont.)

1. $A \leftarrow \emptyset$
 2. **for** each vertex $v \in V$
 3. **do** $\text{MAKE-SET}(v)$
 4. sort E into non-decreasing order by w
 5. **for** each (u, v) taken from the sorted list $\leftarrow O(E)$
 6. **do if** $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
 7. **then** $A \leftarrow A \cup \{(u, v)\}$
 8. $\text{UNION}(u, v)$
 9. **return** A
- Running time: $O(V+E\lg E+E\lg V)=O(E\lg E)$
- Since $E=O(V^2)$, we have $\lg E=O(2\lg V)=O(\lg V)$

Kruskal's Algorithm

- Kruskal's algorithm is a “**greedy**” algorithm
- Kruskal's greedy strategy produces a globally optimum solution
- Proof for generic approach applies to Kruskal's algorithm too



MAXIMUM FLOW

Max-Flow Min-Cut Theorem (Ford Fulkerson's Algorithm)

Ajit Kumar Behera,
CSE/SIT

Directed Graph Applications

- Shortest Path Problem
 - (Shortest path from one point to another)
- Max Flow problems
 - (Maximum material flow through conduit)
 - Liquid flow through pipes
 - Parts through assembly line
 - Current through electrical network, etc

What is Network Flow ?

- A material coursing through a system from a **source**, where the material is produced to a **sink** where it is consumed.
- The source is **produces** the material at some steady rate, and the sink **consumes** the material at **same rate**.
- The **flow** of material at any point of time in the system is the rate at which the material moves.

What is Network Flow ?

Flow network is a directed graph $G = (V, E)$ such that each edge has a non-negative capacity $c(u,v) \geq 0$.

Two distinguished vertices exist in G namely :

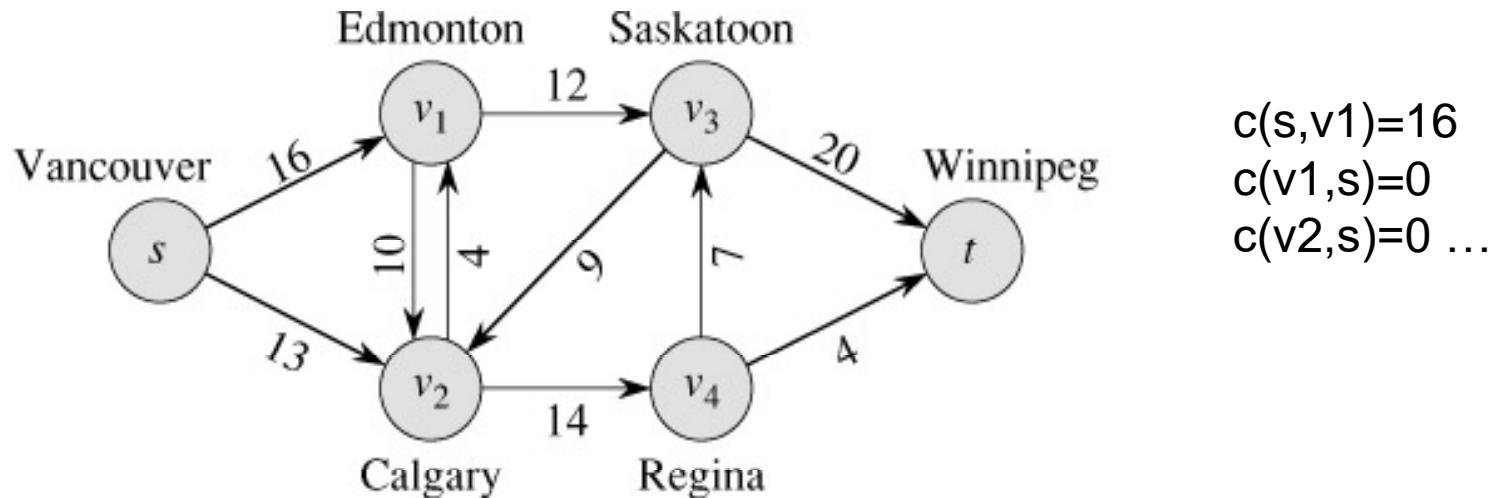
- Source (denoted by s) : In-degree of this vertex is 0.
- Sink (denoted by t) : Out-degree of this vertex is 0.

Flow in a network is an integer-valued function f defined on the edges of G satisfying $0 \leq f(u,v) \leq c(u,v)$, for every edge (u,v) in E .

What is Network Flow ?

- Each edge (u,v) has a non-negative capacity $c(u,v)$.
- If $(u,v) \notin E$ assume $c(u,v)=0$.
- We have source s and sink t .
- Assume that every vertex v in V is on some path from s to t .

Following is an illustration of a network flow:



Conditions for Network Flow

For each edge $(u,v) \in E$, the flow $f(u,v)$ is a real valued function that must satisfy 3 conditions :

- Capacity Constraint : $\forall u,v \in V, f(u,v) \leq c(u,v)$
- Skew Symmetry : $\forall u,v \in V, f(u,v) = -f(v,u)$
- Flow Conservation: $\forall u \in V - \{s,t\}, \sum_{v \in V} f(s,v) = 0$

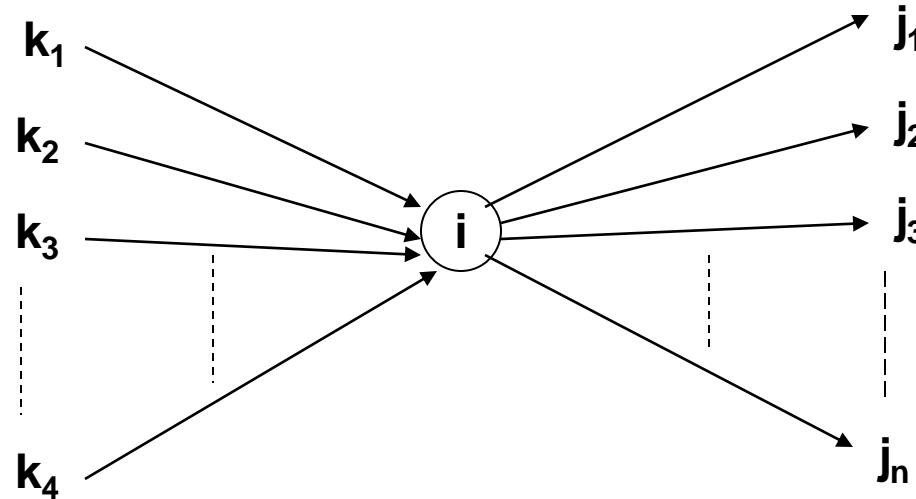
Skew symmetry condition implies that $f(u,u)=0$.

Conditions for Network Flow

3. Flow conservation constraint :

Total net flow at vertex must equal 0.

$$\sum_j f(i, j) - \sum_k f(k, i) = 0 \text{ for all } i \in V - \{s, t\}$$



flow in equals flow out

The Value of a Flow.

The value of a flow is given by :

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$$

- The flow into the node is same as flow going out from the node and thus the flow is **conserved**.
- The total amount of flow from source s = total amount of flow into the sink t .

Example of a flow

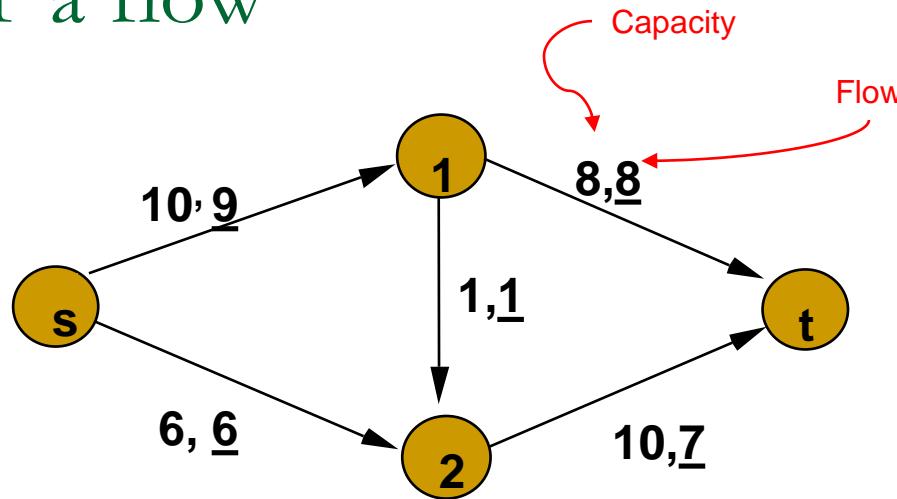


Table illustrating Flows and Capacity across different edges of graph above:

$$f_{s,1} = 9, c_{s,1} = 10 \text{ (Valid flow since } 10 > 9\text{)}$$

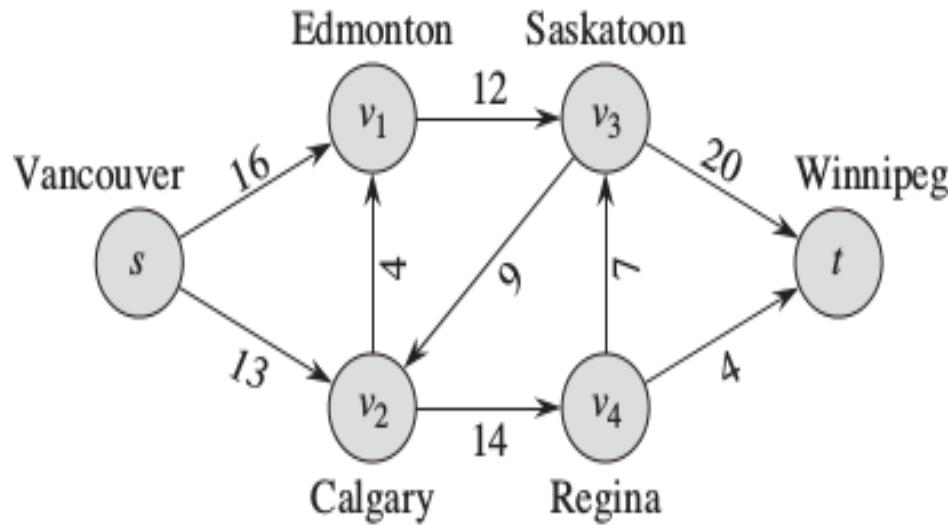
$$f_{s,2} = 6, c_{s,2} = 6 \text{ (Valid flow since } 6 \geq 6\text{)}$$

$$f_{1,t} = 8, c_{1,t} = 8 \text{ (Valid flow since } 8 \geq 8\text{)}$$

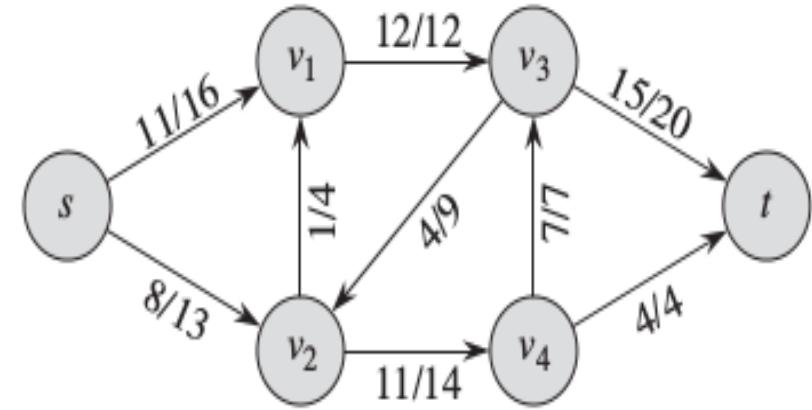
$$f_{2,t} = 7, c_{2,t} = 10 \text{ (Valid flow since } 10 > 7\text{)}$$

The flow across nodes 1 and 2 are also conserved as **flow into them = flow out**.

Maximum Flow



(a)

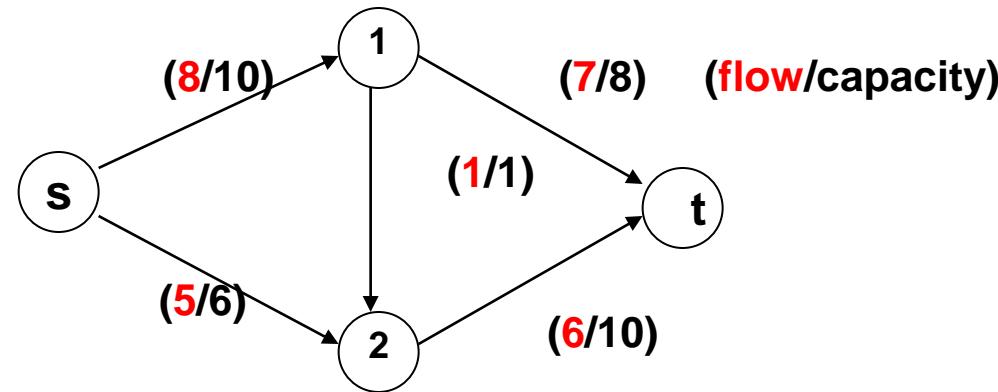


(b)

(a) A flow network $G = (V, E)$ and (b) A flow f in G with value $|f|=19$. Each edge (u,v) is labeled by $f(u,v)/c(u,v)$ (represent flow and capacity)

Maximum Flow

- We refer a *flow f* as maximum if it is feasible and maximize $\sum_k f(s, k)$. where $f(s, k)$ is flow out of source s .
- Problem:



- Objective: To find a maximum *flow f*

The Ford-Fulkerson Algorithm for Max Flow

- Key ingredients:-
 - Residual Networks
 - Augmenting Paths
 - Cut
- Limitations :-
 - Flow should be integral or rational
 - On each iteration residual capacity should be integral

The Residual Network

- Given a flow network and a flow, the residual network consists of edges that can admit more flow.
- Suppose we have a flow network $G = (V, E)$ with source s and sink t .
- Let f be a flow in G and consider a pair of vertices $u, v \in V$.
- The amount of additional flow that can be pushed from u to v before exceeding capacity $c(u, v)$ is the **residual capacity** of (u, v) given by:

$$c_f(u, v) = c(u, v) - f(u, v)$$

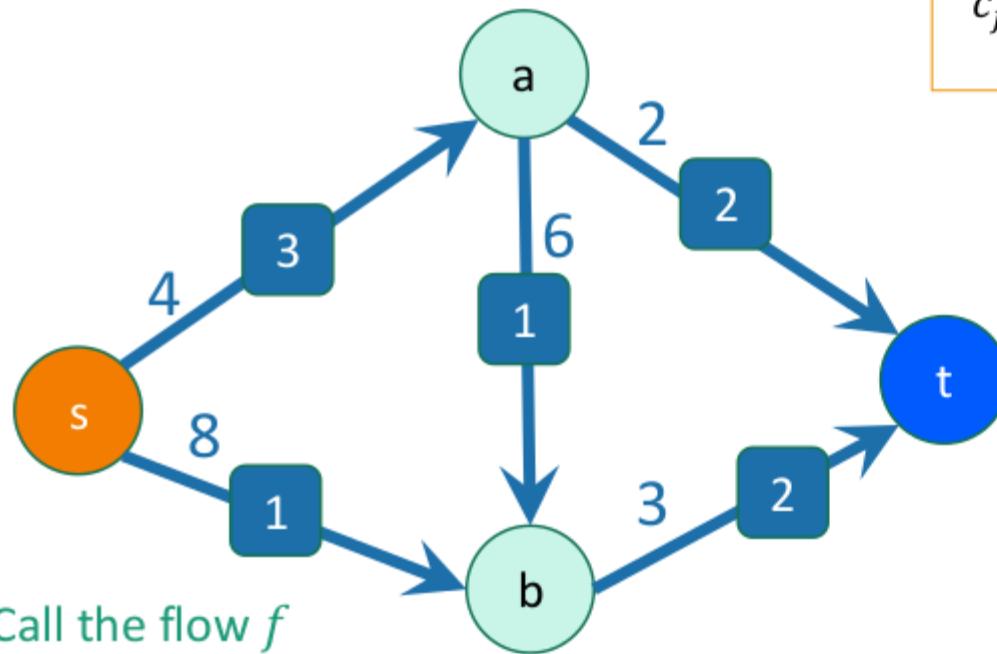
Cont...

Our assumption:

For $G = (V, E)$, if $(u, v) \in E \Rightarrow (v, u) \notin E$

Tool: Residual networks

Say we have a flow

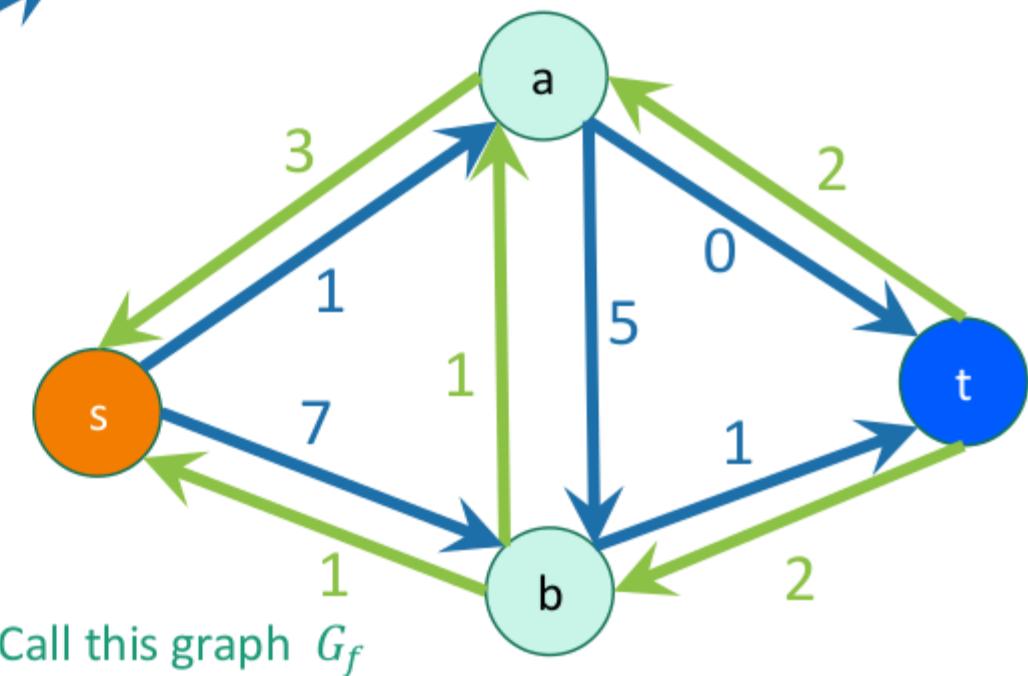


Create a new **residual network** from this flow:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{else} \end{cases}$$

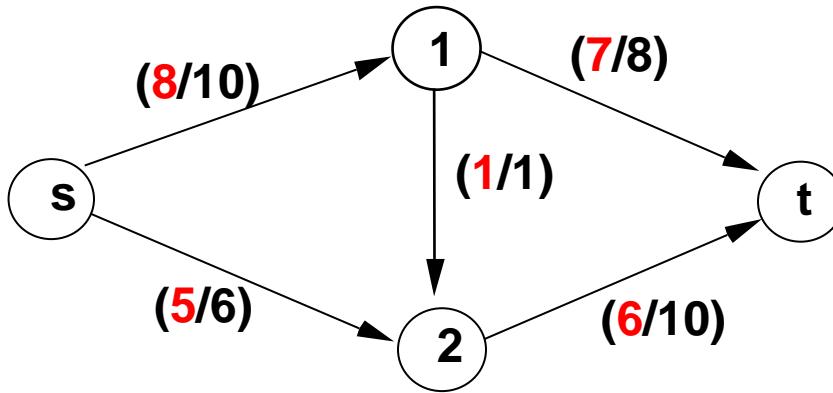
- $f(u, v)$ is the flow on edge (u, v) .
- $c(u, v)$ is the capacity on edge (u, v) .

Forward edges are the amount that's left.
Backwards edges are the amount that's been used.



Residual Network (Example)

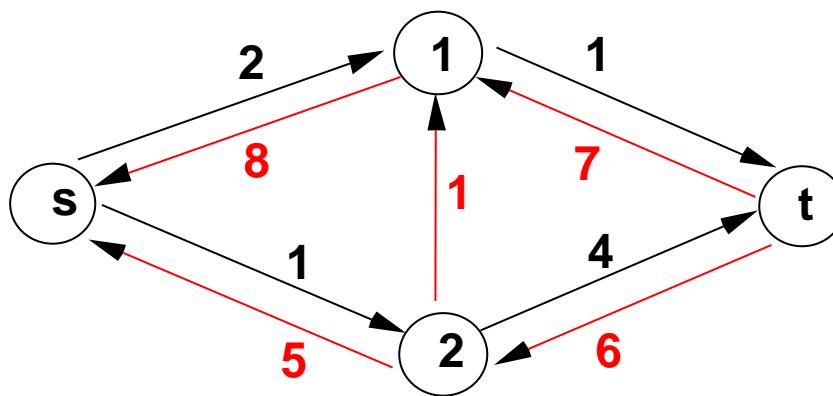
Flow Network



$$\begin{array}{c} \text{(flow / capacity)} \\ f(u, v) / c(u, v) \end{array}$$

A diagram illustrating a single directed edge between nodes u and v . The edge is labeled with the fraction $f(u, v) / c(u, v)$, where $f(u, v)$ is the current flow and $c(u, v)$ is the total capacity.

Residual Network



Residual capacity $c_f(u, v)$

$$\begin{array}{c} c(u, v) - f(u, v) \\ f(u, v) \end{array}$$

A diagram illustrating the calculation of residual capacity. It shows two parallel edges between nodes u and v . The top edge has a capacity $c(u, v)$ and a flow $f(u, v)$. The bottom edge has a capacity $c(u, v) - f(u, v)$ and a flow $f(u, v)$.

Why do we care about residual networks?

Lemma:

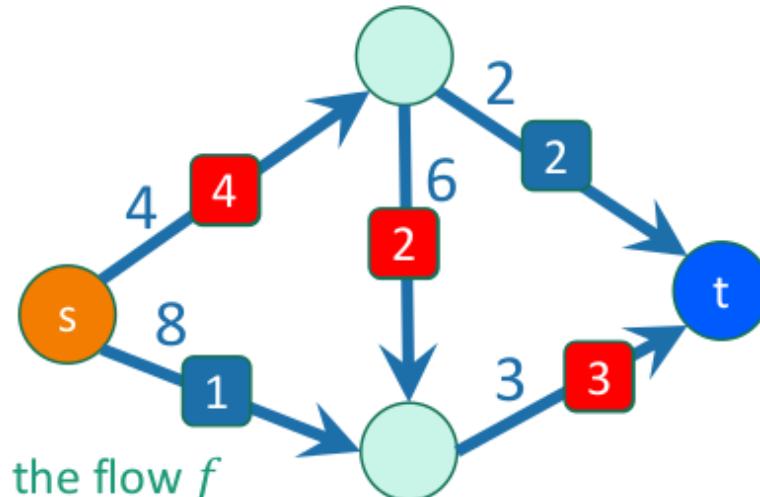
- t is not reachable from s in $G_f \Leftrightarrow f$ is a max flow.

Example:

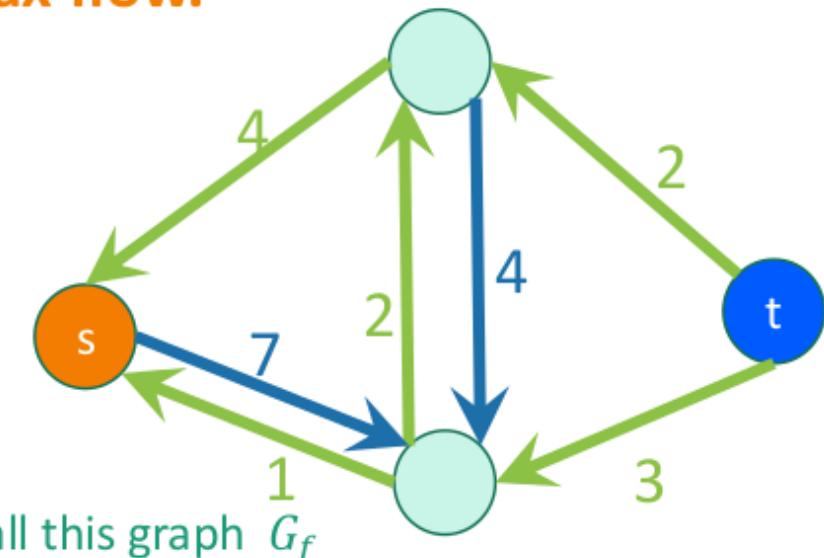
Now we get this residual graph:

Now we can't reach t from s .

So the lemma says that f is a max flow.



Call the flow f
Call the graph G

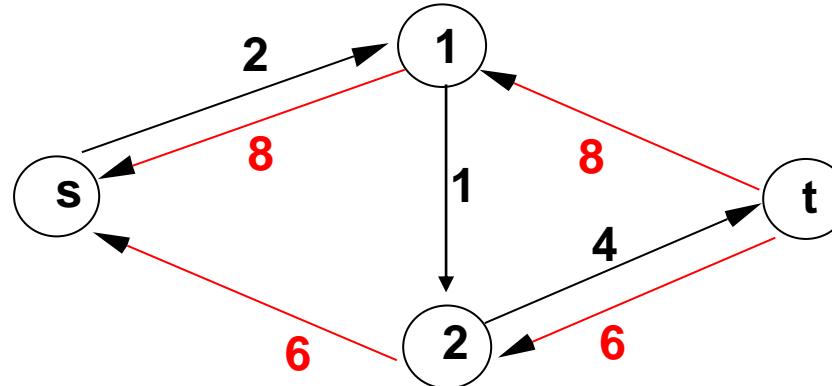
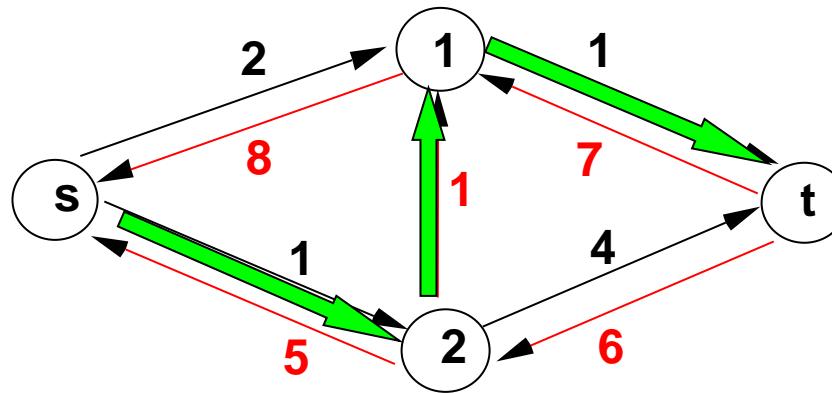


Call this graph G_f

Augmenting Paths

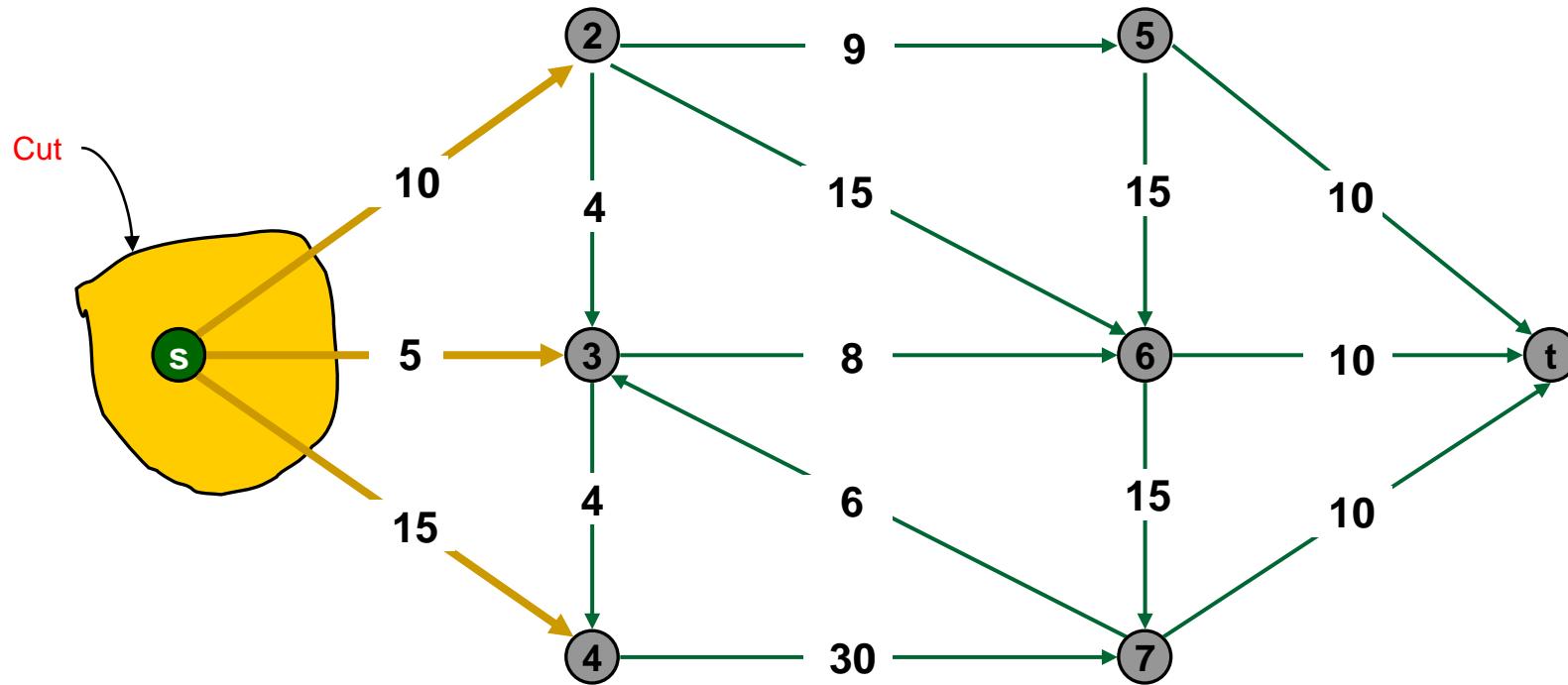
- Given a flow network $G = (V, E)$ and a flow f , an **augmenting path p** is a simple path from s to t in the **residual network G_f** .
- We can put more flow from s to t through p
- **Claim:-**
If there is an augmenting path, then we can **increase** the flow along the path.

Augmenting Paths



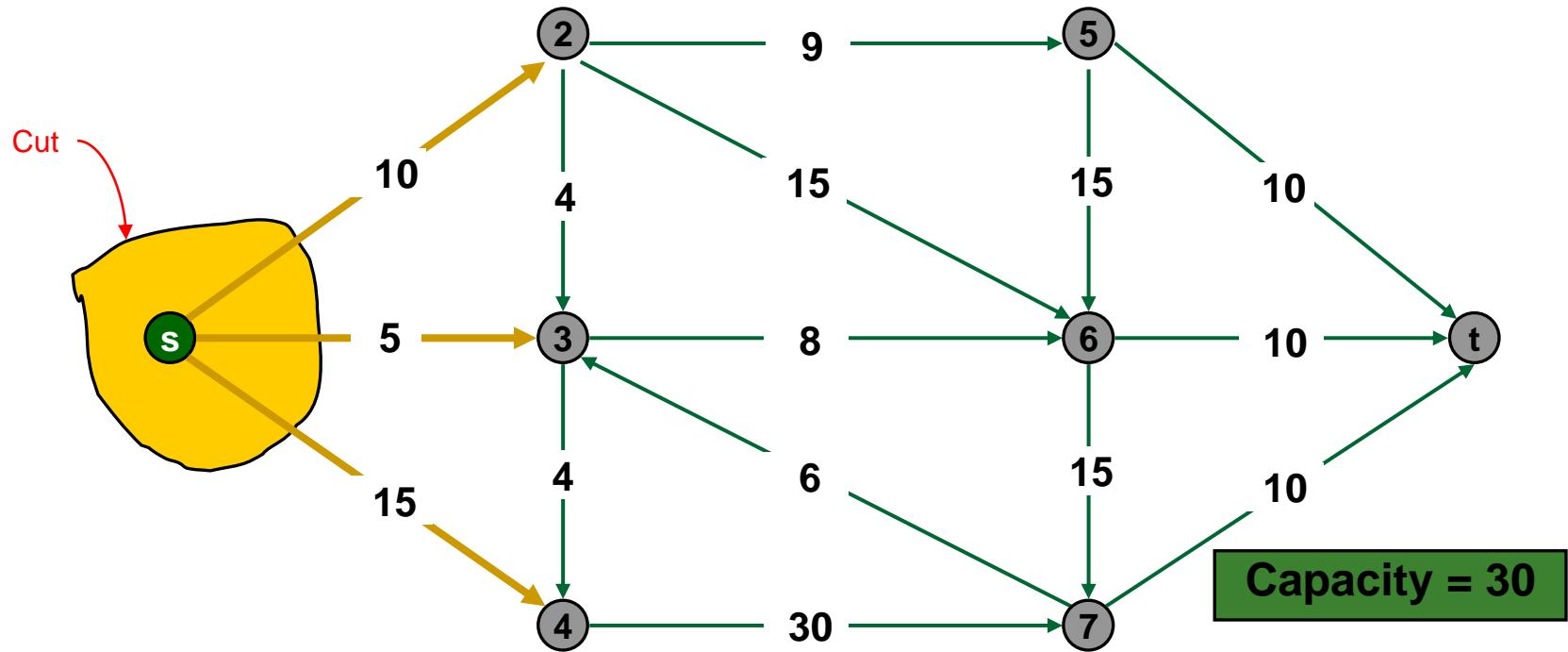
Cuts of Flow Networks

A **Cut** in a network is a partition of V into S and T ($T=V-S$) such that s (source) is in S and t (target) is in T .



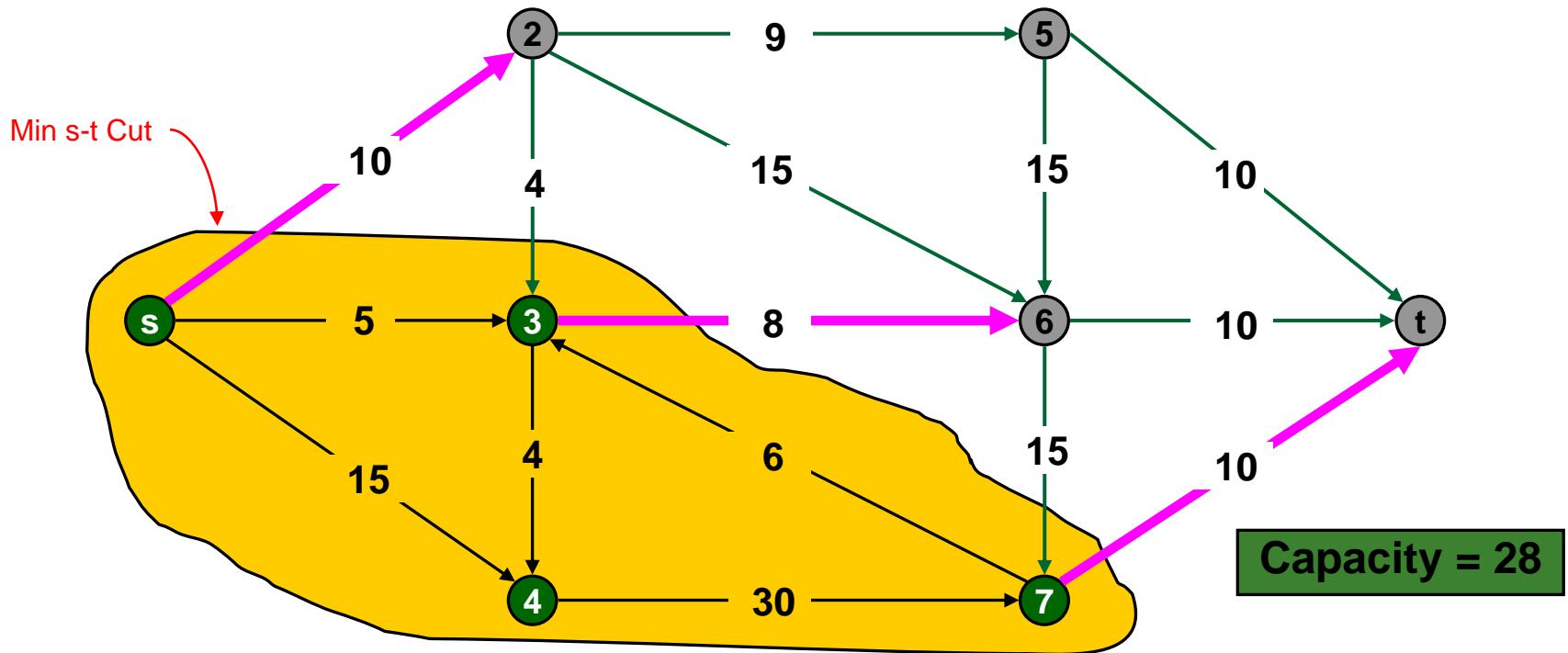
Capacity of Cut (S, T)

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$



Min Cut

Min s-t cut (Also called as a Min Cut) is a cut of **minimum capacity**



Methods

Max-Flow Min-Cut Theorem

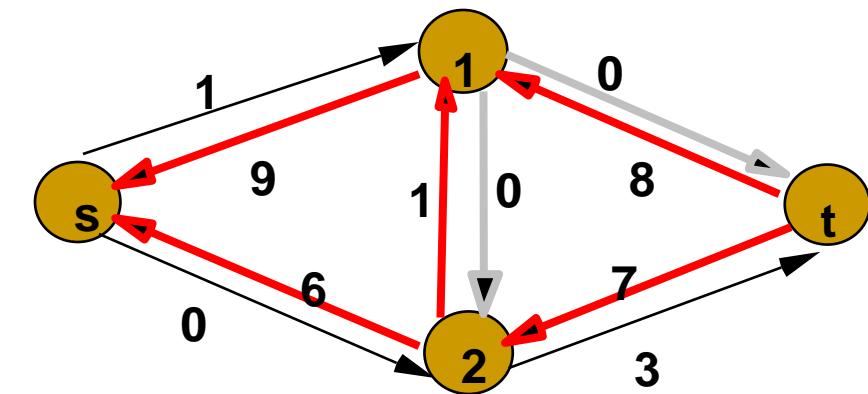
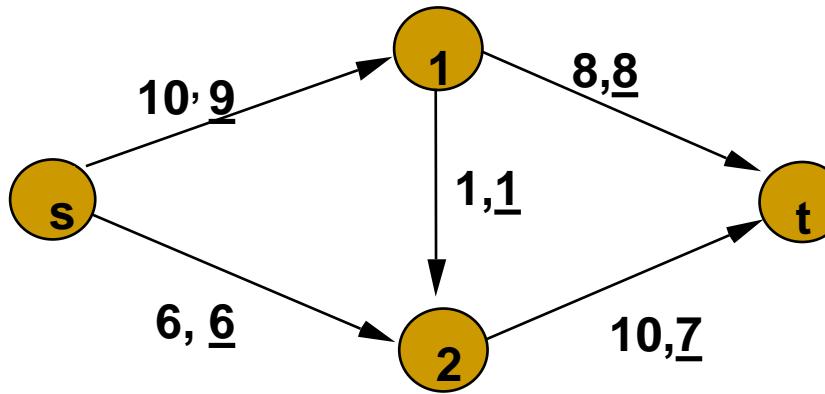
The Ford-Fulkerson Method

The Ford-Fulkerson Method

- Try to improve the flow, until we reach the maximum value of the flow
- The residual capacity of the network with a flow f is given by:

The residual capacity c_f of an edge (i,j) equals $c(i,j) - f(i,j)$ when (i,j) is a forward edge, and equals $f(i,j)$ when (i,j) is a backward edge. Moreover the residual capacity of an edge is always non-negative.

$$c_f(u, v) = c(u, v) - f(u, v)$$



Original Network

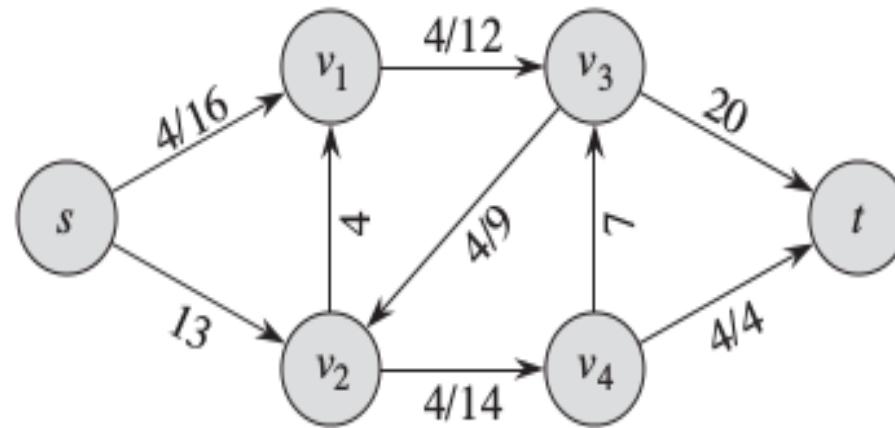
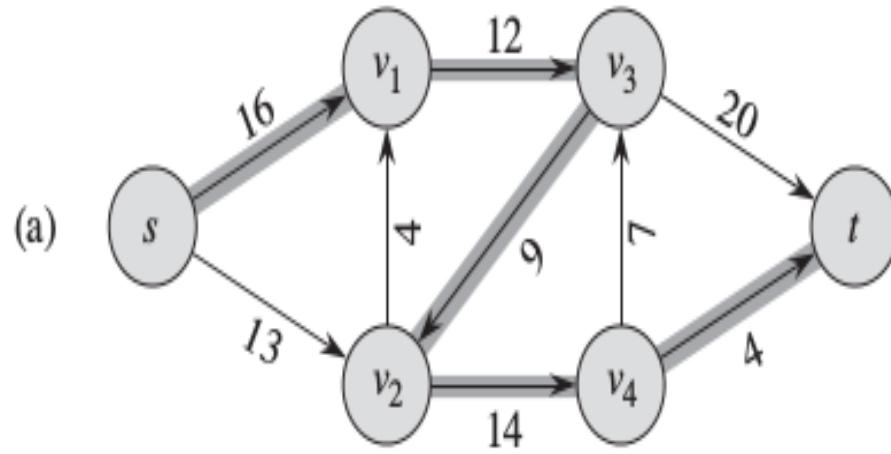
Residual Network

The Ford-Fulkerson Method

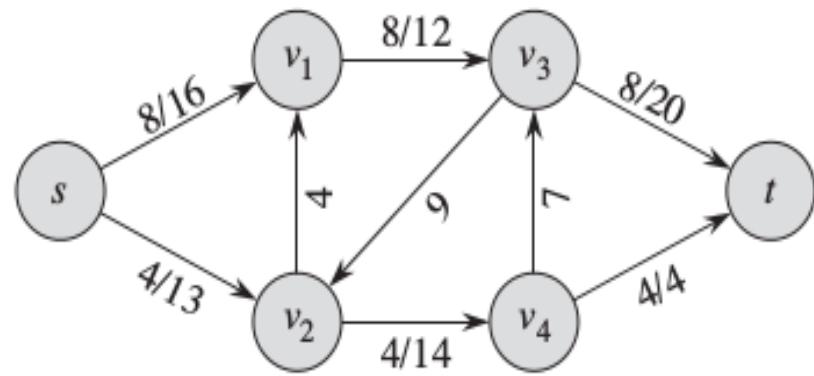
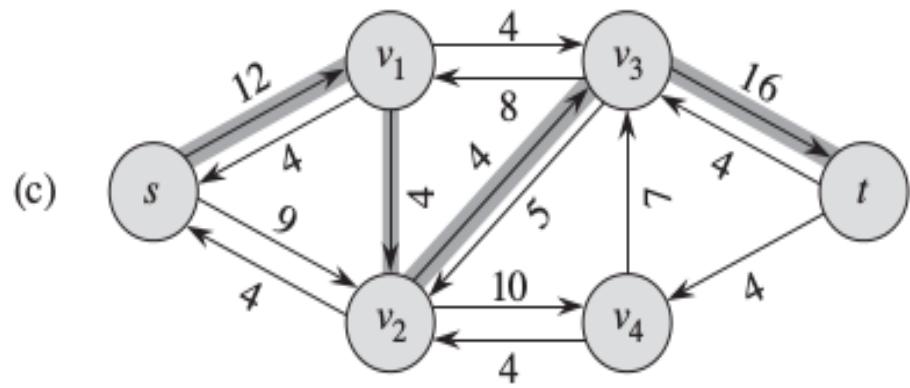
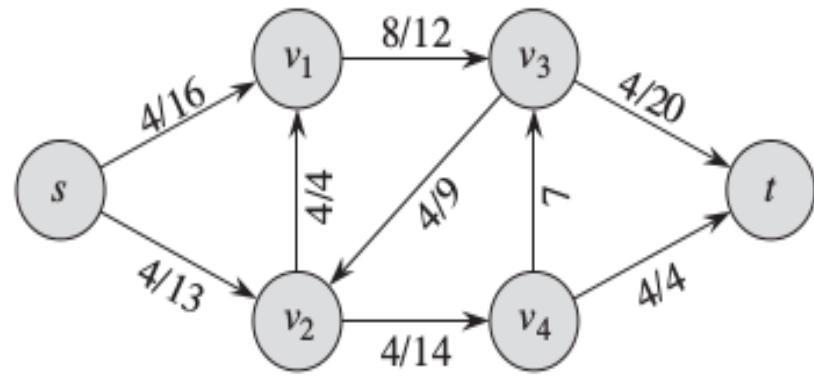
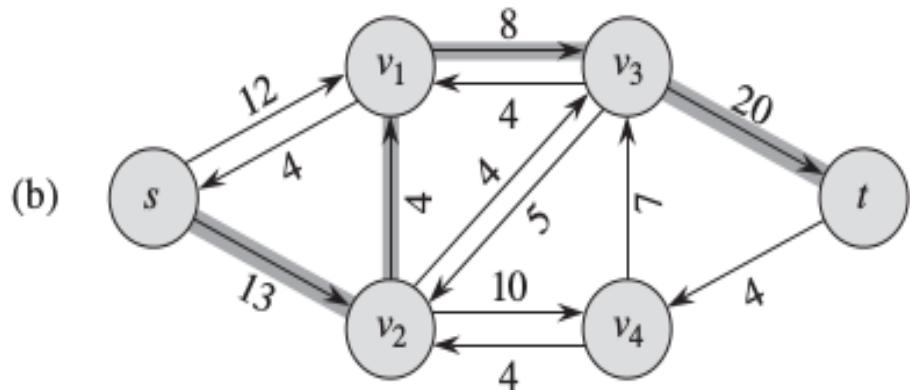
FORD-FULKERSON(G, s, t)

```
1  for each edge  $(u, v) \in G.E$            // Initialize flow  $f$  to 0
2     $(u, v).f = 0$ 
3  while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ 
4     $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$ 
5    for each edge  $(u, v)$  in  $p$ 
6      if  $(u, v) \in E$ 
7         $(u, v).f = (u, v).f + c_f(p)$ 
8      else  $(v, u).f = (v, u).f - c_f(p)$ 
```

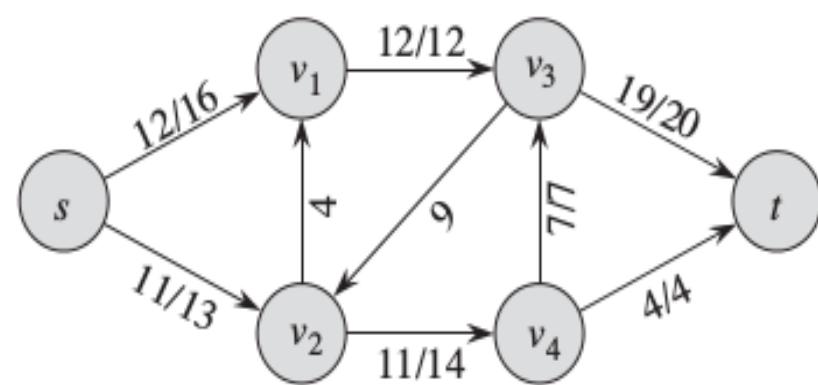
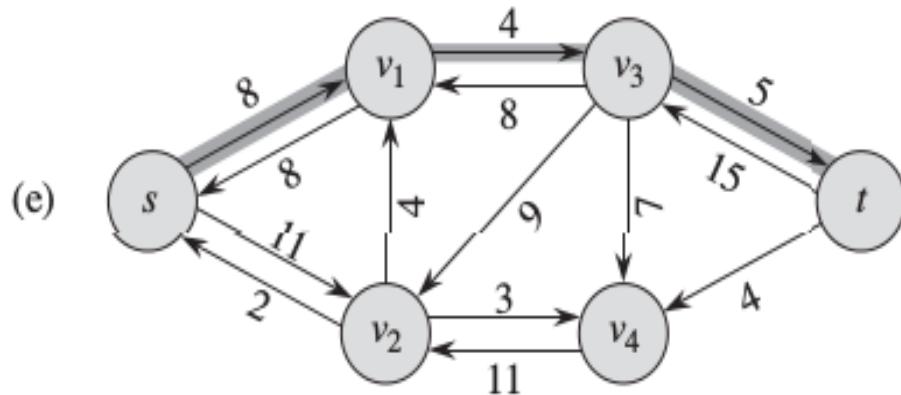
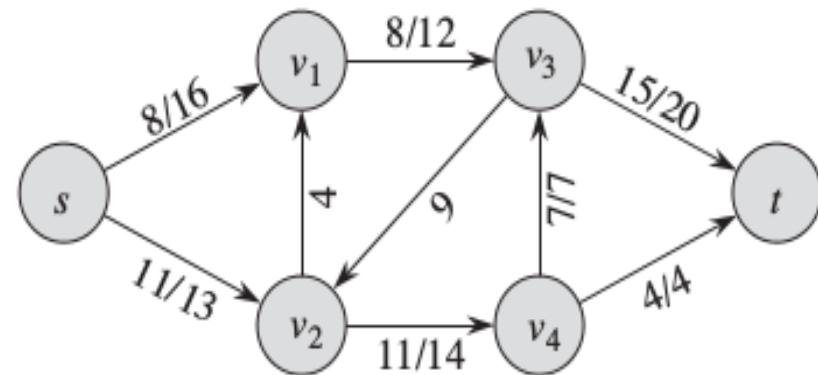
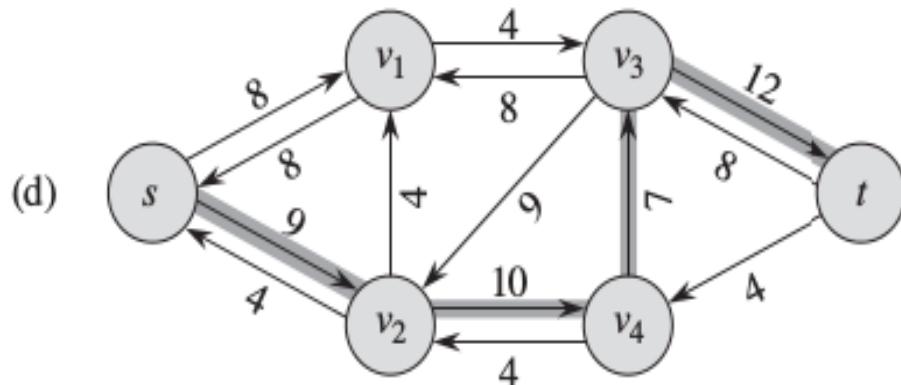
Example-1



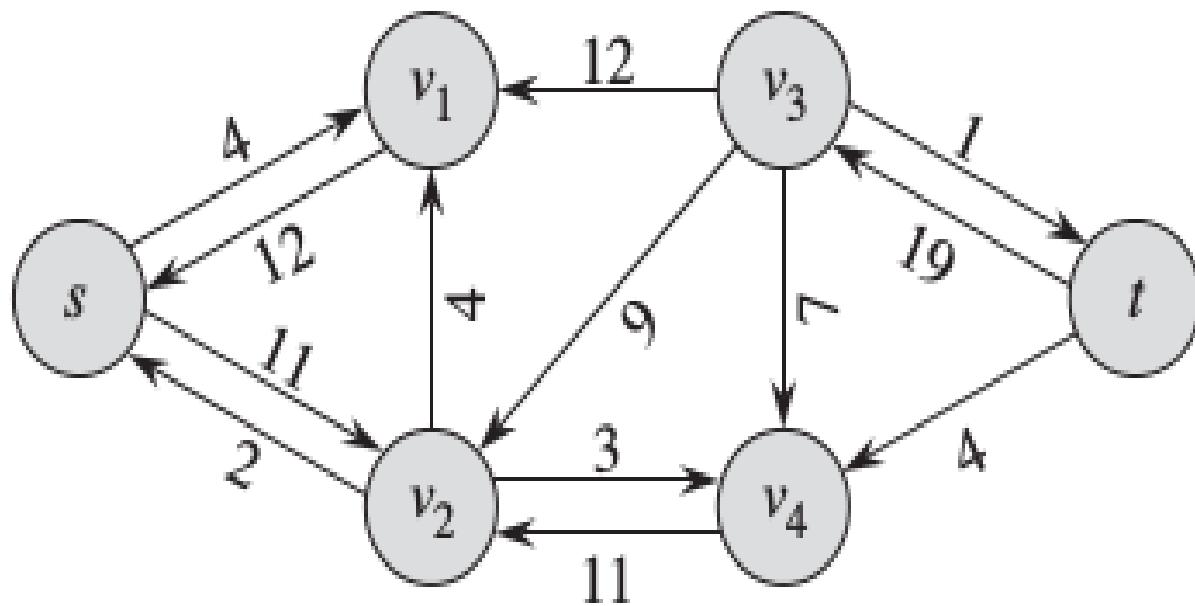
Cont...



Cont...



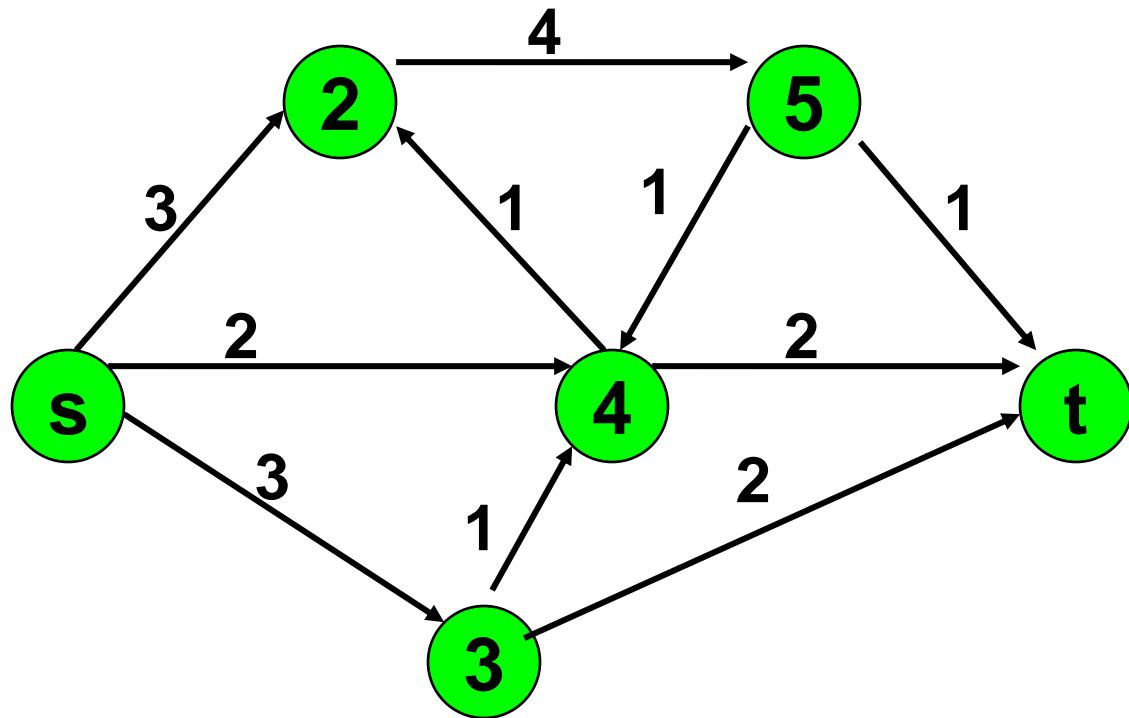
(f)



(f) The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

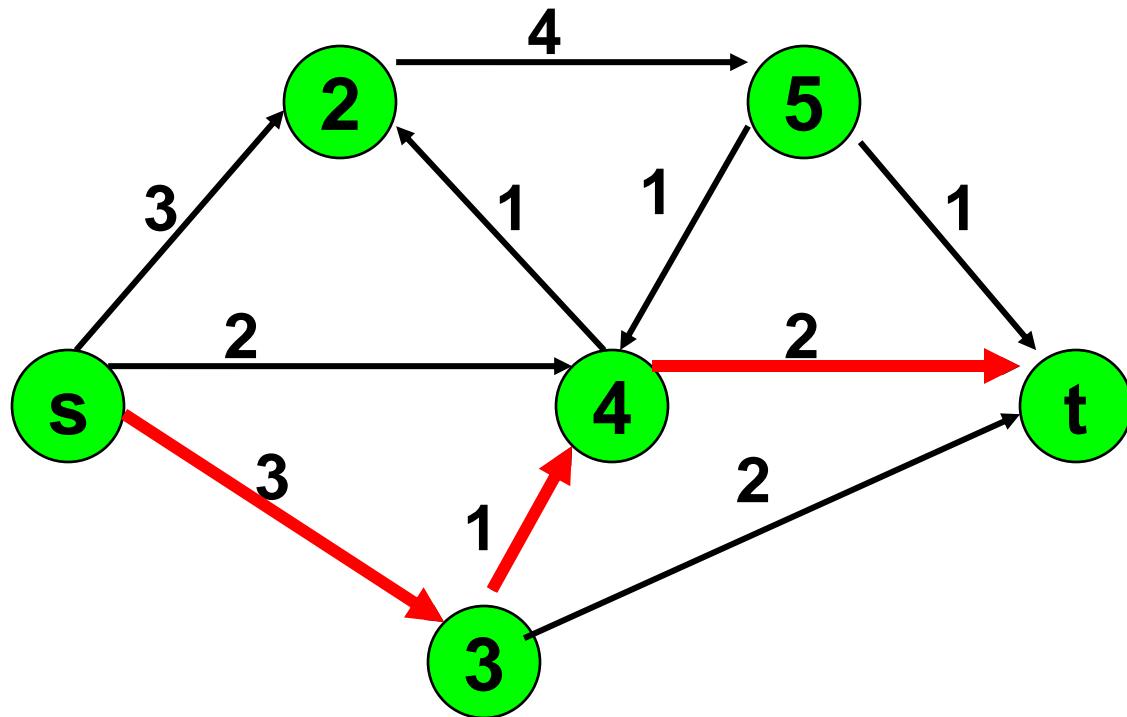
Ex. 2

Ford-Fulkerson Max Flow



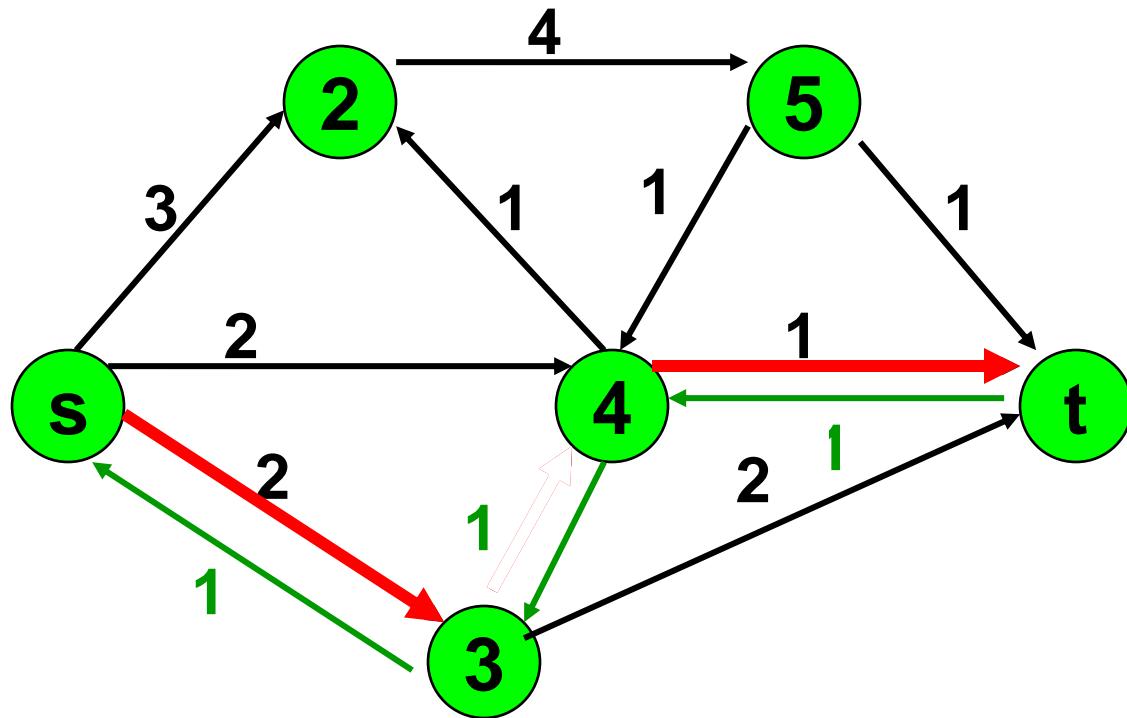
This is the original network,
and the original residual
network.

Ford-Fulkerson Max Flow



Find any s - t path in $G(x)$

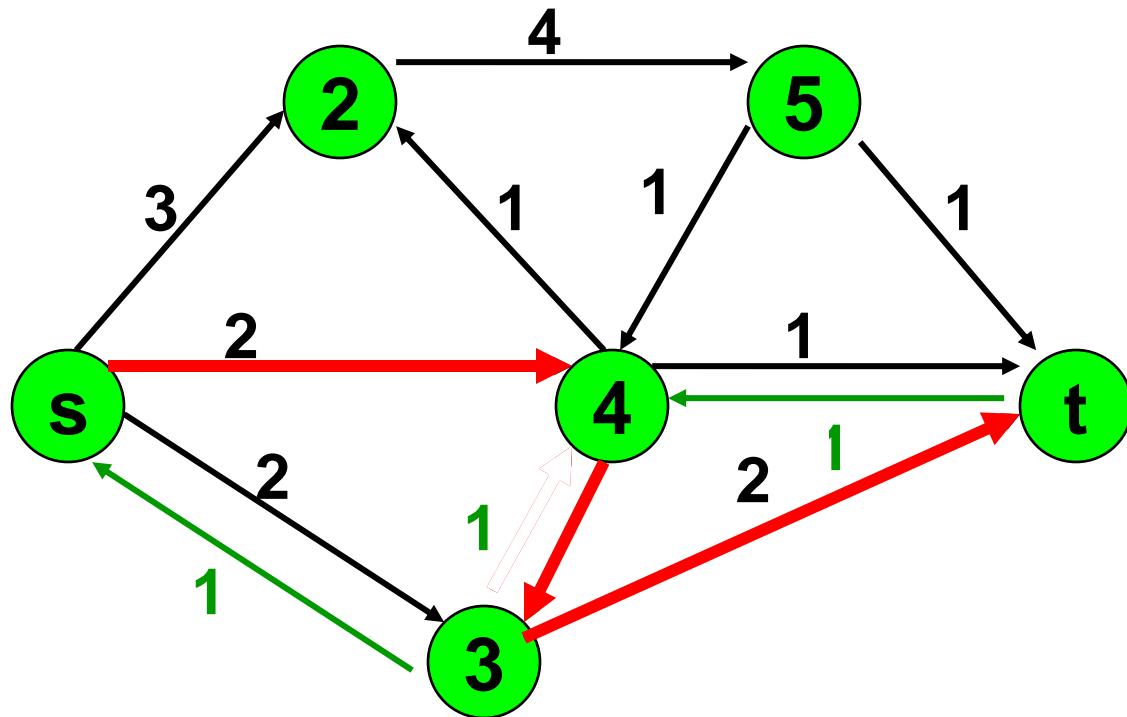
Ford-Fulkerson Max Flow



Determine the capacity Δ of the path.

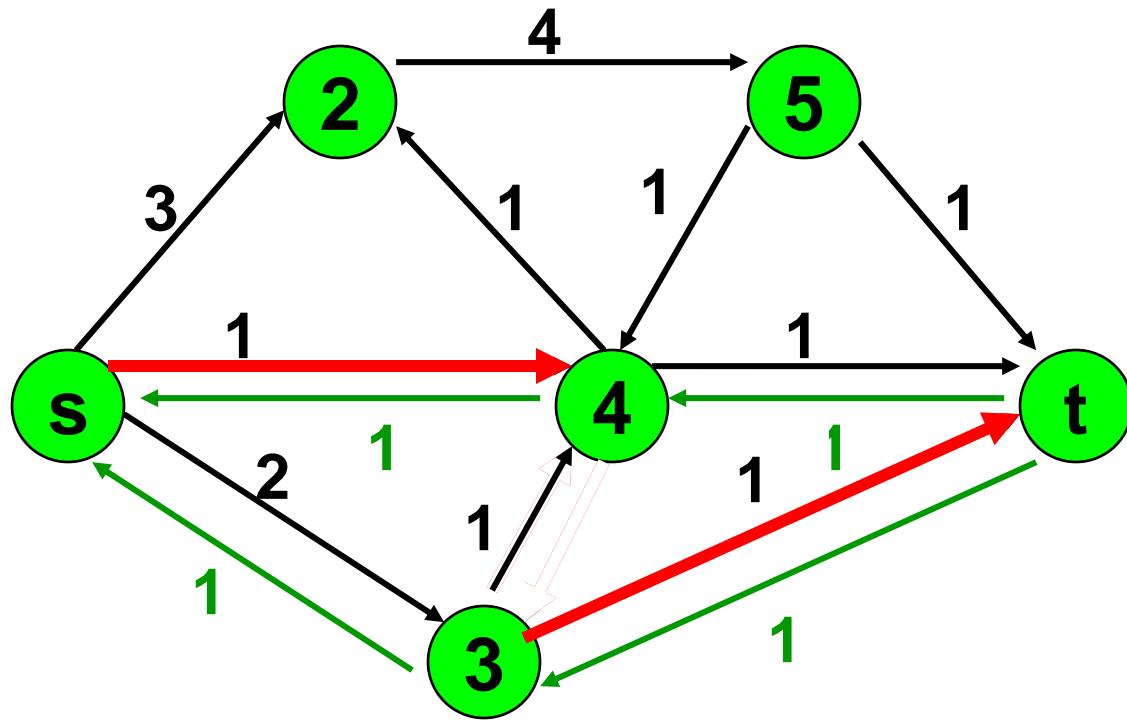
Send Δ units of flow in the path.
Update residual capacities.

Ford-Fulkerson Max Flow



Find any s - t path

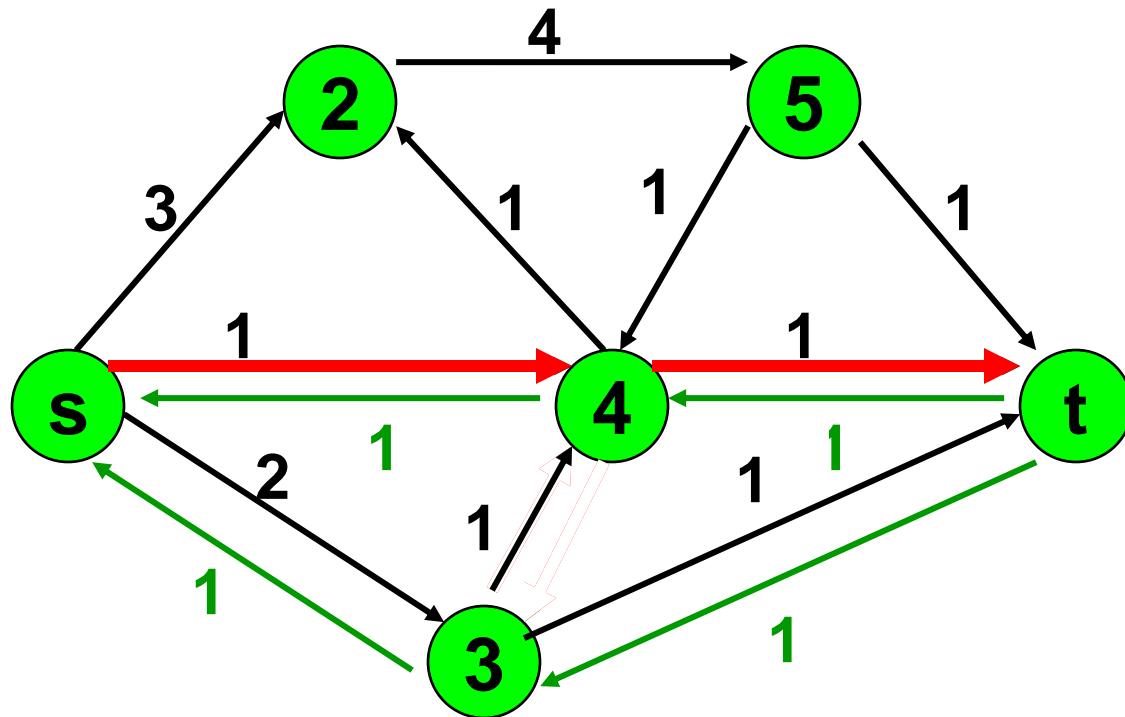
Ford-Fulkerson Max Flow



Determine the capacity Δ of the path.

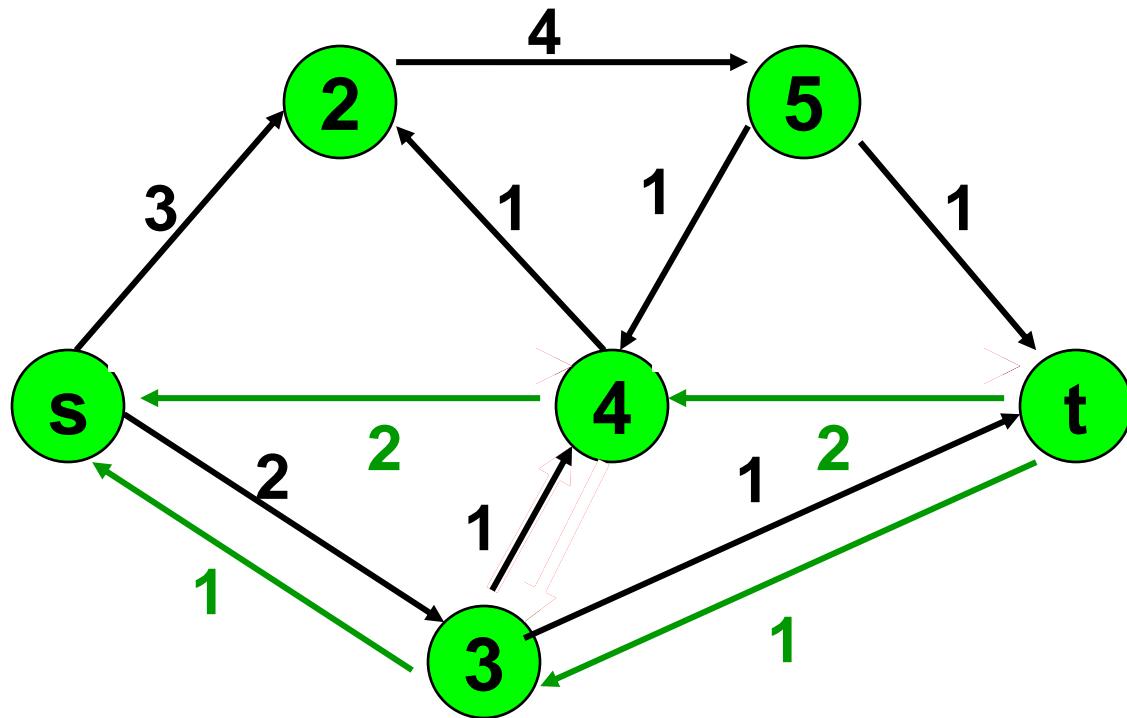
Send Δ units of flow in the path.
Update residual capacities.

Ford-Fulkerson Max Flow



Find any s - t path

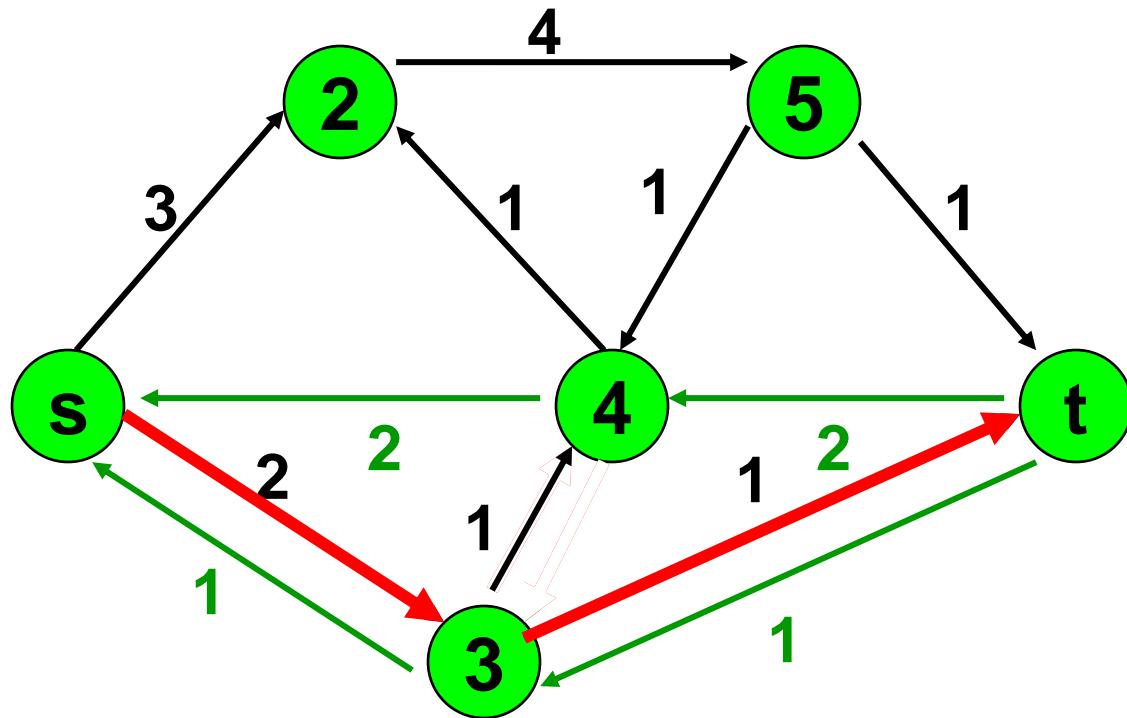
Ford-Fulkerson Max Flow



Determine the capacity Δ of the path.

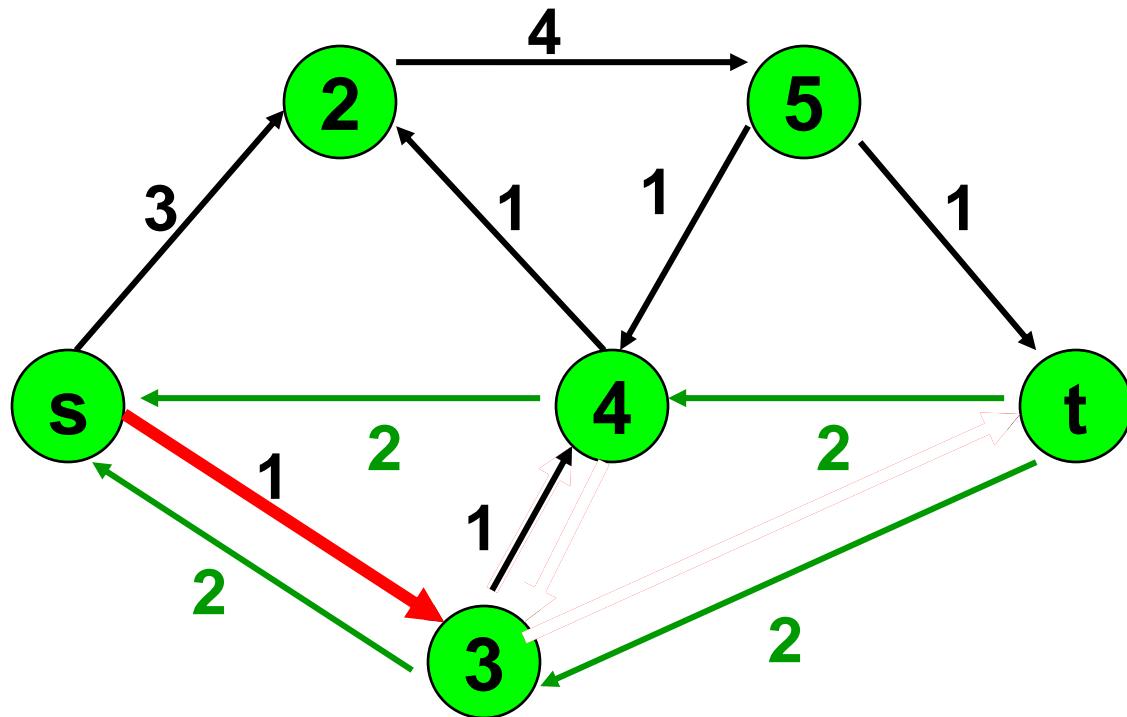
Send Δ units of flow in the path.
Update residual capacities.

Ford-Fulkerson Max Flow



Find any s - t path

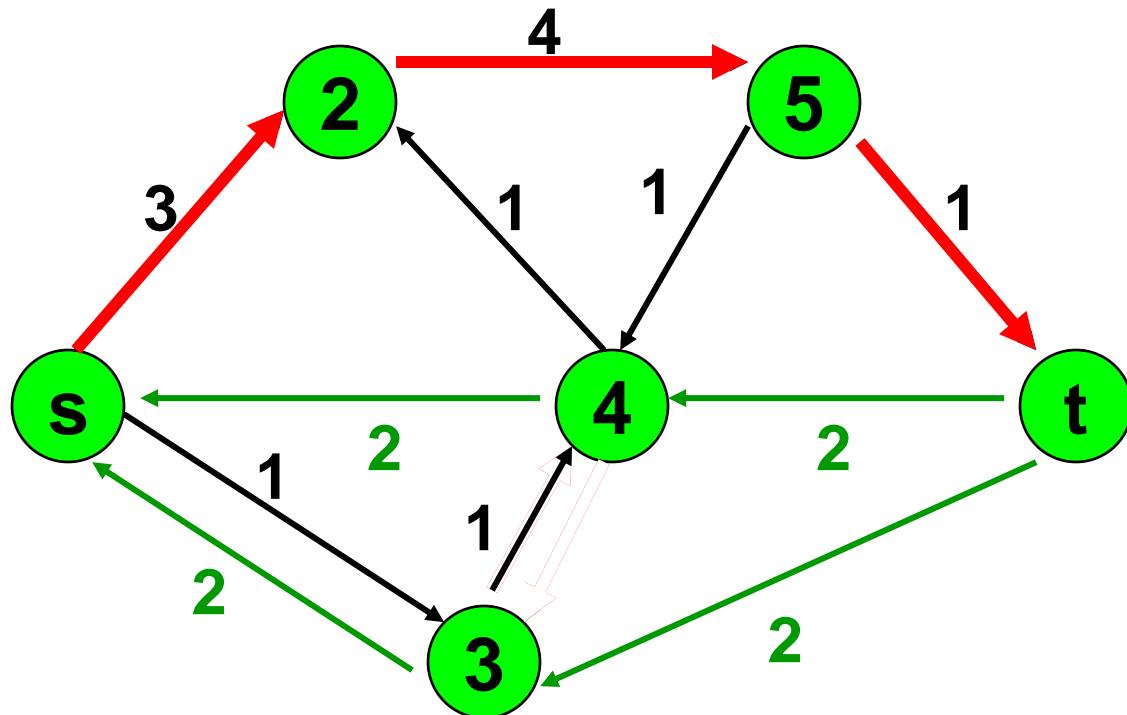
Ford-Fulkerson Max Flow



Determine the capacity Δ of the path.

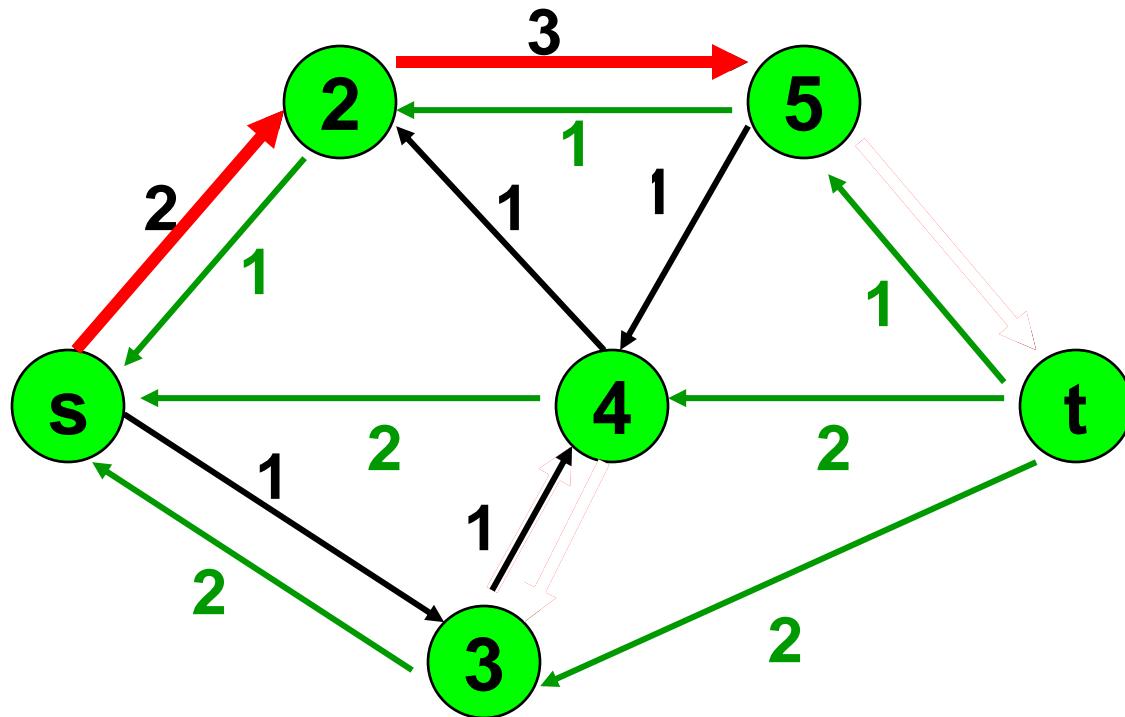
Send Δ units of flow in the path.
Update residual capacities.

Ford-Fulkerson Max Flow



Find any s - t path

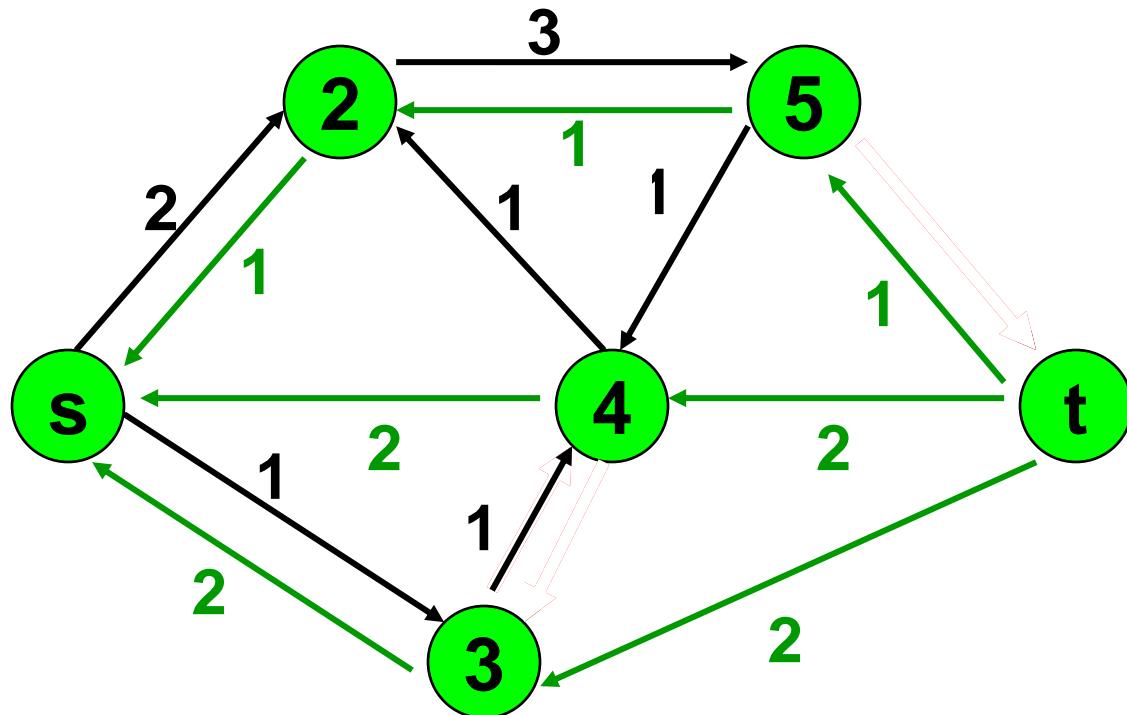
Ford-Fulkerson Max Flow



Determine the capacity Δ of the path.

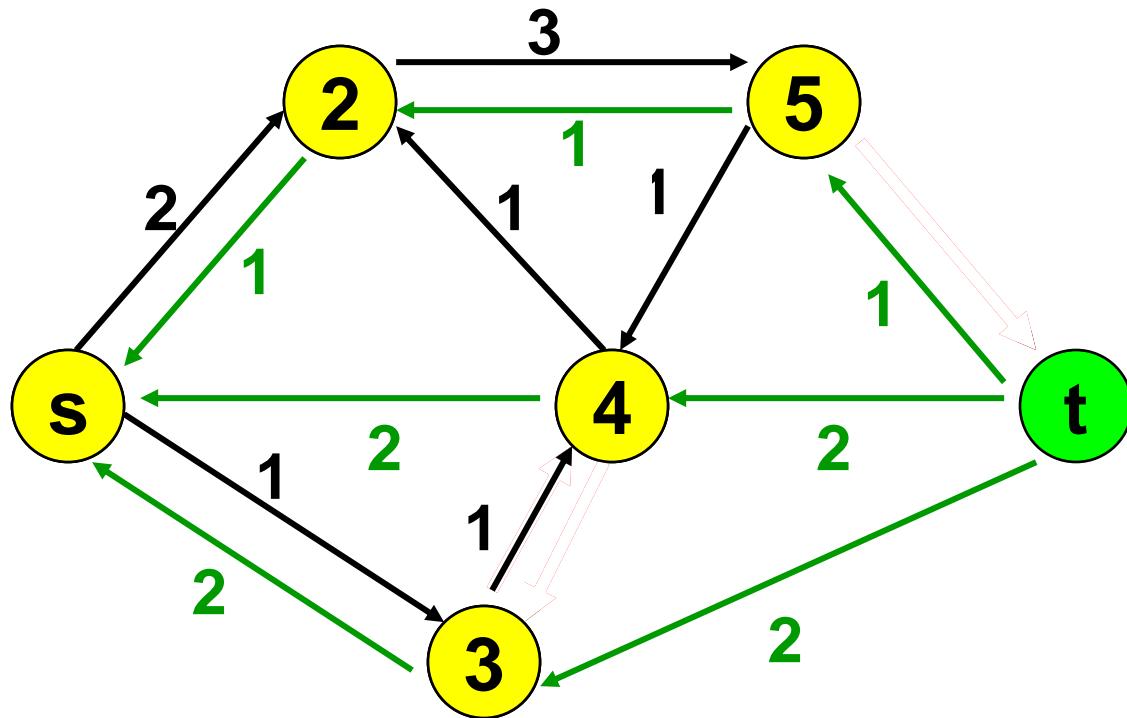
Send Δ units of flow in the path.
Update residual capacities.

Ford-Fulkerson Max Flow



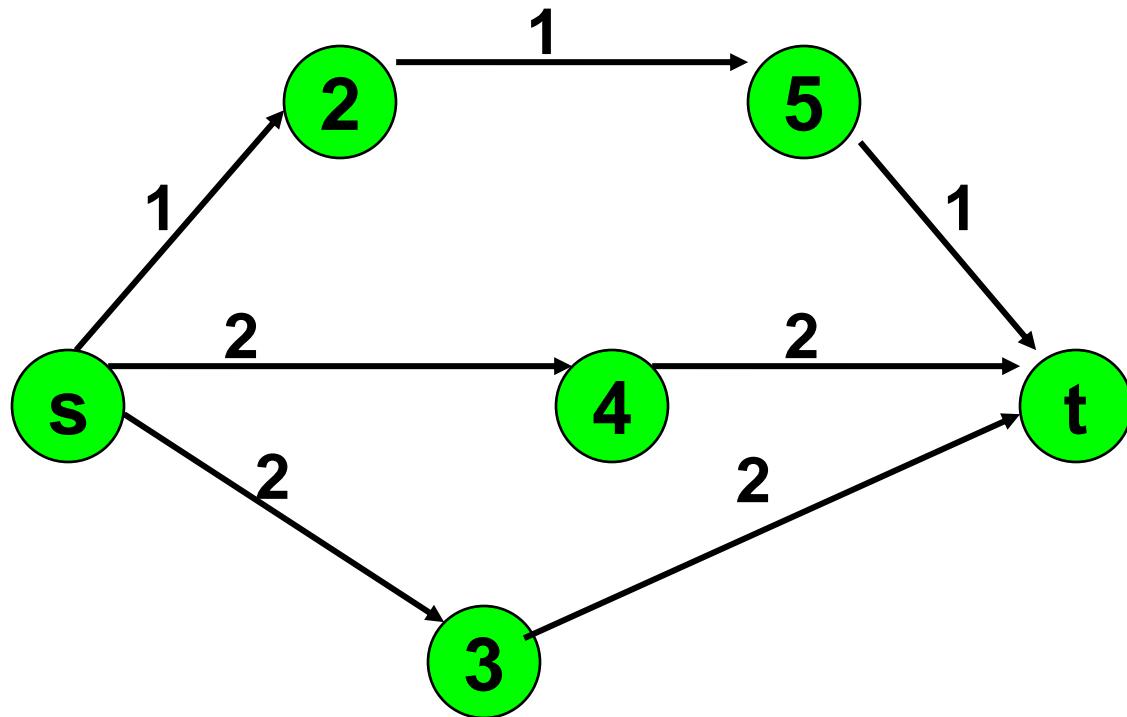
There is no s-t path in the residual network. This flow is optimal

Ford-Fulkerson Max Flow



These are the nodes that are reachable from node s .

Ford-Fulkerson Max Flow



Here is the optimal flow

Time Complexity

Line -3: The time to find path in a residual network is $O(V+E') = O(E)$ { we can apply BFS or DFS }.

If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD -FULKERSON executes the while loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

So, the time complexity is $O(E | f^* |)$

NP Completeness (Module V)

Jasaswi Prasad Mohanty

Problem Types

- The algorithms we have studied so far have been ***polynomial-time algorithms***: on inputs of size n , their worst-case running time is $O(n^k)$ for some constant k .
- Question: ***Whether all problems can be solved in polynomial time?***
 - Answer: No
 - There are problems, such as Turing's famous “Halting Problem,” that cannot be solved by any computer, no matter how much time we allow.
 - There are also problems that can be solved, but not in time $O(n^k)$ for any constant k .
- Problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require super-polynomial time as being intractable, or hard.

NP-Complete Problem

- No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.
- We can say the status of “NP-complete” problems, is unknown.

Class P

- The class P consists of those problems that are solvable in polynomial time.
- These problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- **Example:** LCS Finding, Sorting

Class NP

- The class NP consists of those problems that are “verifiable” in polynomial time.
- **Question: What is verifiable?**
 - If we were somehow given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.
- **Example:**
 - In the hamiltonian cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, \dots, v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time if the sequence is a hamiltonian cycle or not.
 - Any problem in P is also in NP since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. So $P \subseteq NP$.

Examples of Polynomial time Problem vs NP-Complete Problem

- **Shortest vs. Longest simple paths:**

- We can find *shortest paths from a single source in a directed graph* $G = (V, E)$ in $O(VE)$ time.
- Finding a *longest simple path between two vertices* is difficult. Determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

- **Euler tour vs. hamiltonian cycle:**

- An *Euler tour of a connected, directed graph* $G = (V, E)$ is a cycle that traverses each *edge of G exactly once, although it is allowed to visit each vertex more than once*. We can find an Euler tour in only $O(E)$ time.
- A *hamiltonian cycle* of a directed graph $G = (V, E)$ is a simple cycle that contains each *vertex in V*. Determining whether a directed graph has a hamiltonian cycle is NP-complete.

Class NP-Complete

- A problem is in NPC if it is in NP and is as “hard” as any problem in NP.
- *If any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial time algorithm.*
- Most theoretical computer scientists believe that the NP-complete problems are intractable.
- If we can establish a problem as NP-complete, then we would try to spend our time in developing an approximation algorithm or solve for a tractable special case, rather than searching for a fast algorithm that solves the problem exactly.

Abstract Problem

- An *abstract problem* Q to be a binary relation on a set I of problem instances and a set S of problem solutions.
- **Example:** an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices.
 - A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.
 - The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices.

Abstract Decision Problem

- An abstract decision problem is a function that maps the instance set I to the solution set $\{0, 1\}$.
- **Example:** If $I = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then $\text{PATH}(i) = 1$ (yes) if a shortest path from u to v has at most k edges, and $\text{PATH}(i) = 0$ (no) otherwise.
- Many abstract problems are not decision problems, but rather *optimization problems, which require some value to be minimized or maximized.*
- We can recast an optimization problem as a decision problem easily.

Encoding

- An *encoding of a set S* of abstract objects is a mapping e from S to the set of binary strings.
- Example: $e(17) = 10001$
- We call a problem whose instance set is the set of binary strings a *concrete problem*.
- A concrete problem is *polynomial-time solvable*, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k .
- *Formal definition of Complexity Class P*: It is the set of concrete decision problems that are polynomial-time solvable.
- Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \rightarrow \{0, 1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$.

Polynomial-time computable

- We say that a function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$.

Formal-Language Framework

- An **alphabet** Σ is a finite set of symbols.
- A **language L over Σ** is any set of strings made up of symbols from Σ .
- **Example:**
 - If $\Sigma = \{0,1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$ is the language of binary representation of prime numbers.
- We denote the *empty string* by \in , the *empty language* by ϕ , and the *language of all strings* over Σ by Σ^* .
- **Example:**
 - If $\Sigma = \{0,1\}$, the set $\Sigma^* = \{\in, 0, 1, 00, 10, 01, 11, 000, \dots\}$ is the set of all binary strings.

Formal-Language Framework

- We can perform a variety of operations on languages such as ***union, intersection, complement, and concatenation.***
- We define the ***complement of L*** by $\bar{L} = \Sigma^* - L$.
- ***The concatenation L_1L_2 of two languages L_1 and L_2*** is the language

$$L = \{x_1x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$$

- The ***closure or Kleene star of a language L is the language***

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup L^4 \cup \dots$$

where L^k is the language obtained by concatenating L to it self k times.

Formal-Language Framework

- The decision problem PATH has the corresponding language:

$\text{PATH} = \{\langle G, u, v, k \rangle : G = (V, E) \text{ is an undirected graph,}$
 $u, v \in V,$
 $k \geq 0 \text{ is an integer, and}$
 $\text{there exists a path from } u \text{ to } v \text{ in } G$
 $\text{consisting of at most } k \text{ edges}\} .$

Relation between Decision problems and algorithms

- An algorithm A *accepts a string* $x \in \{0, 1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1.
- The language *accepted by an algorithm A* is the set of strings $L = \{x \in \{0, 1\}^*: A(x) = 1\}$, that is, the set of strings that the algorithm accepts.
- An algorithm A *rejects a string* x if $A(x) = 0$.
- Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it.
 - The algorithm may loop forever.
- A language L is *decided by an algorithm A* if every binary string in L is accepted by A and every binary string not in L is rejected by A.

Relation between Decision problems and algorithms

- A language L is ***accepted in polynomial time by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.***
- A language L is ***decided in polynomial time by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0, 1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.***
- To accept a language, an algorithm need only produce an answer when provided a string in L, but to decide a language, it must correctly accept or reject every string in $\{0, 1\}^*$.
- The language PATH can be accepted and decided in polynomial time.
- An alternative definition of the complexity class P:
 $P = \{L \subseteq \{0, 1\}^* : \text{there exists an algorithm A that decides } L \text{ in polynomial time}\}$

Hamiltonian-cycle problem

- A *hamiltonian cycle* of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V .
- A graph that contains a hamiltonian cycle is said to be *hamiltonian*; otherwise, it is *nonhamiltonian*.
- We can define the *hamiltonian-cycle problem*, “Does a graph G have a hamiltonian cycle?” as a formal language:
$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\}$$
- We do not have a polynomial time algorithm to solve hamiltonian-cycle problem.

Verification Algorithm

- We define a ***verification algorithm*** as being a two-argument algorithm A , where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***.
- A two-argument algorithm A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$.
- The language verified by a verification algorithm A is
 $L = \{x \in \{0, 1\}^*: \text{there exists } y \in \{0, 1\}^*, \text{ such that } A(x, y) = 1\}$

Complexity class NP

- The **complexity class NP** is the class of languages that can be verified by a polynomial-time algorithm.
- More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that
 $L = \{x \in \{0, 1\}^*: \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$
- We say that algorithm A **verifies language L in polynomial time**.
- **Example:** HAM-CYCLE \in NP

Relation between P and NP

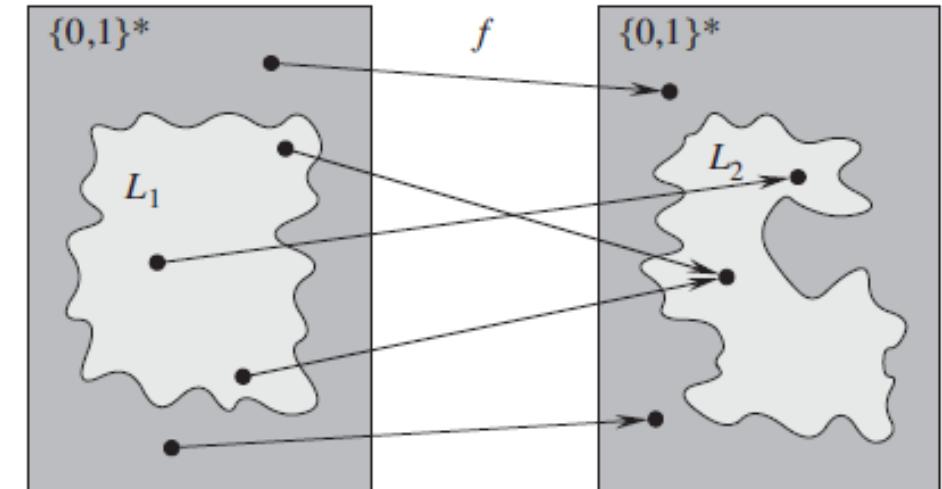
- If $L \in P$, then $L \in NP$, since if there is a polynomial-time algorithm to decide L , the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L .
- Thus, $P \subseteq NP$.
- It is unknown whether $P = NP$, but most researchers believe that P and NP are not the same class.
- Intuitively, the class P consists of problems that can be solved quickly.
- The class NP consists of problems for which a solution can be verified quickly.

Reducibility

- A problem Q can be reduced to another problem Q' if any instance of Q can be “easily rephrased” as an instance of Q', the solution to which provides a solution to the instance of Q.
- **Example:** The problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations.
- Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$.
- Thus, if a problem Q reduces to another problem Q', then Q is, in a sense, “no harder to solve” than Q'.

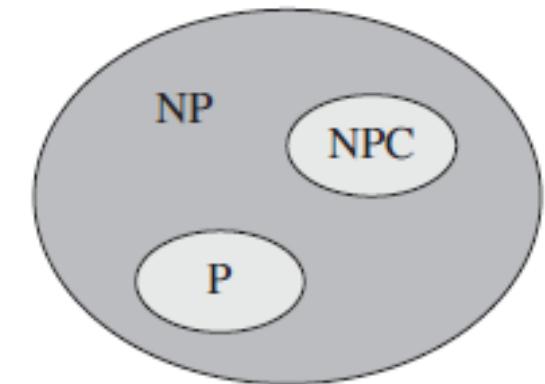
Reducibility

- A language L_1 is **polynomial-time reducible** to a language L_2 , written $L_1 \leq_p L_2$, if there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.
- We call the function f the **reduction function**, and a polynomial-time algorithm F that computes f is a **reduction algorithm**.
- The reduction function f provides a polynomial-time mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$.
- If $L_1, L_2 \subseteq \{0, 1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies $L_1 \in P$.



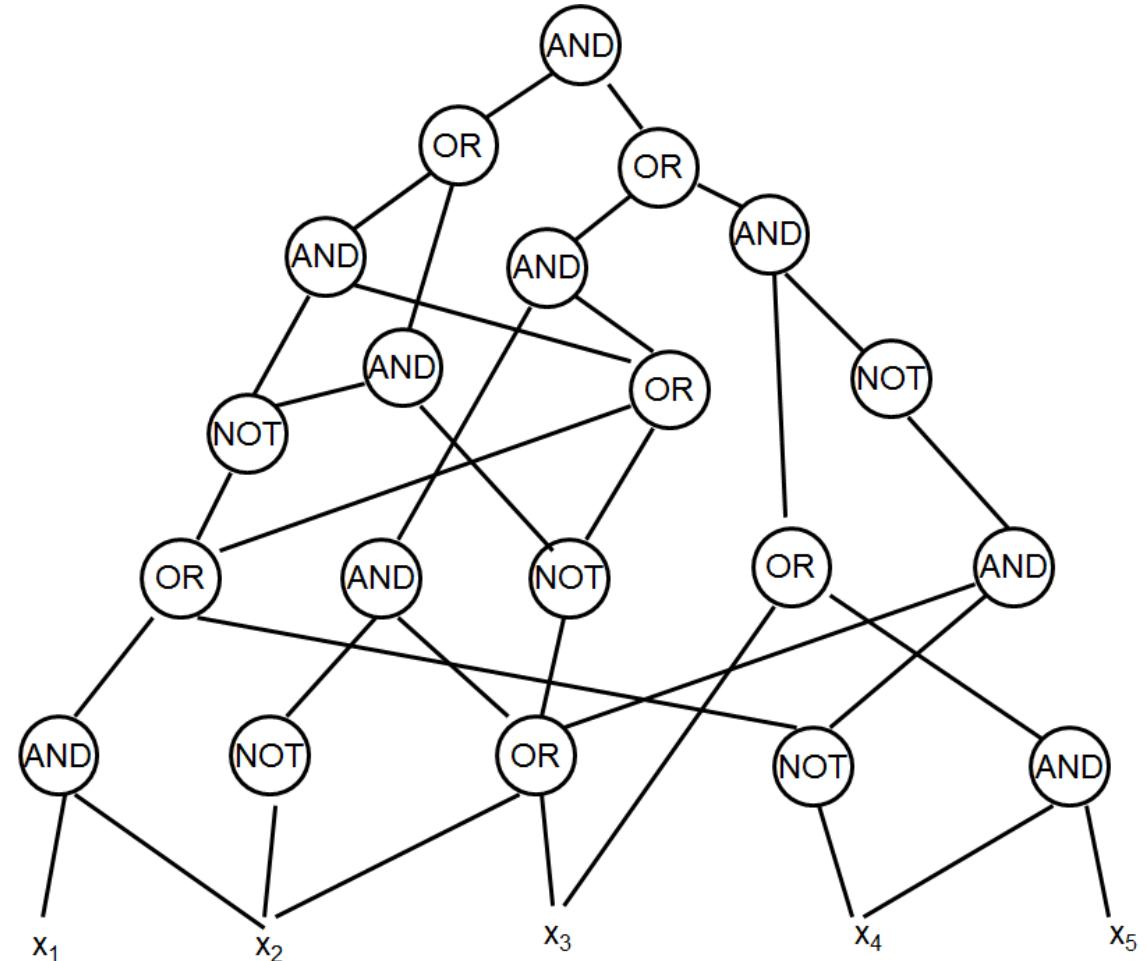
NP-Complete

- A language $L \subseteq \{0, 1\}^*$ is ***NP-complete*** if
 1. $L \in NP$, and
 2. $L' \leq_p L$ for every $L' \in NP$.
- If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***.
- *We also define NPC to be the class of NP-complete languages.*
- ***Theorem:*** If any NP-complete problem is polynomial-time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

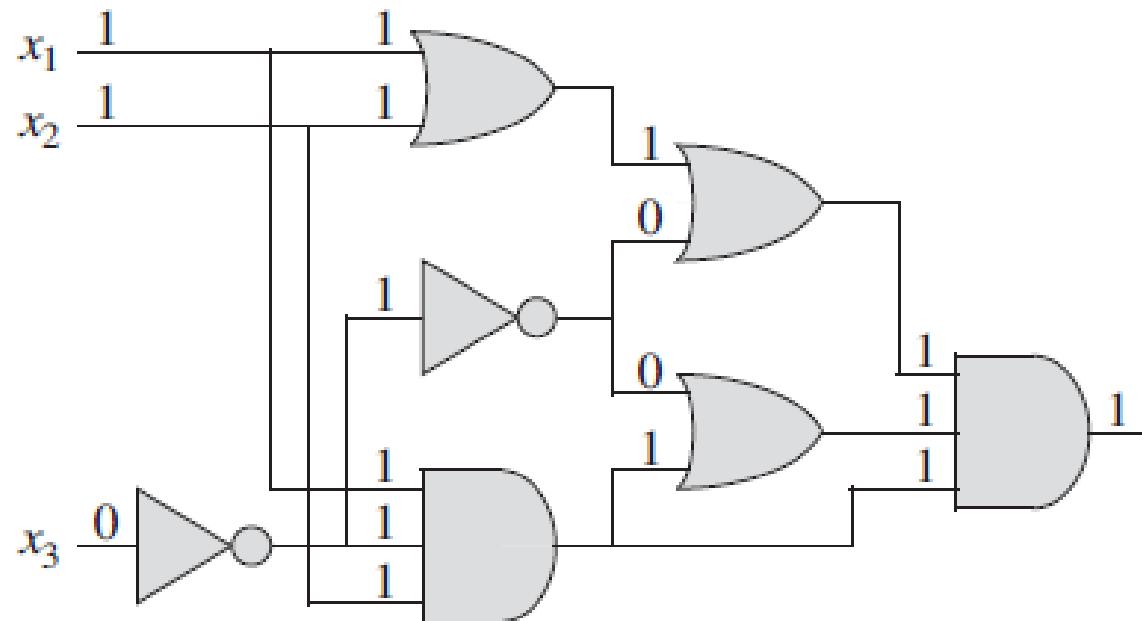


Circuit Satisfiability

- Given a Boolean combination circuit composed of AND, OR & NOT gates, is it satisfiable?
- Formal Language Definition:
 $\text{CIRCUIT-SAT} = \{ \langle C \rangle : C \text{ is a satisfiable combinational circuit} \}$
- This is the first NP-Complete problem.
- This problem is proved as NP-Complete by S.A. Cook in 1971.

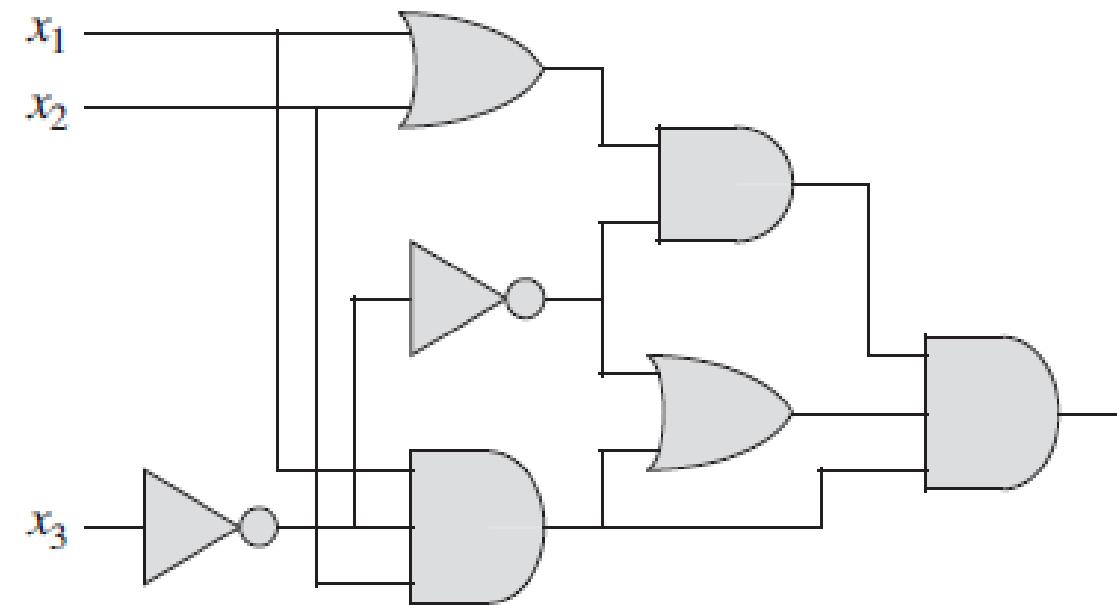


Circuit Satisfiability: Example



Satisfiable Circuit

The assignment $\langle x_1=1, x_2=1, x_3=0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1.



Unsatisfiable Circuit

No assignment to the inputs of this circuit can cause the output of the circuit to be 1.

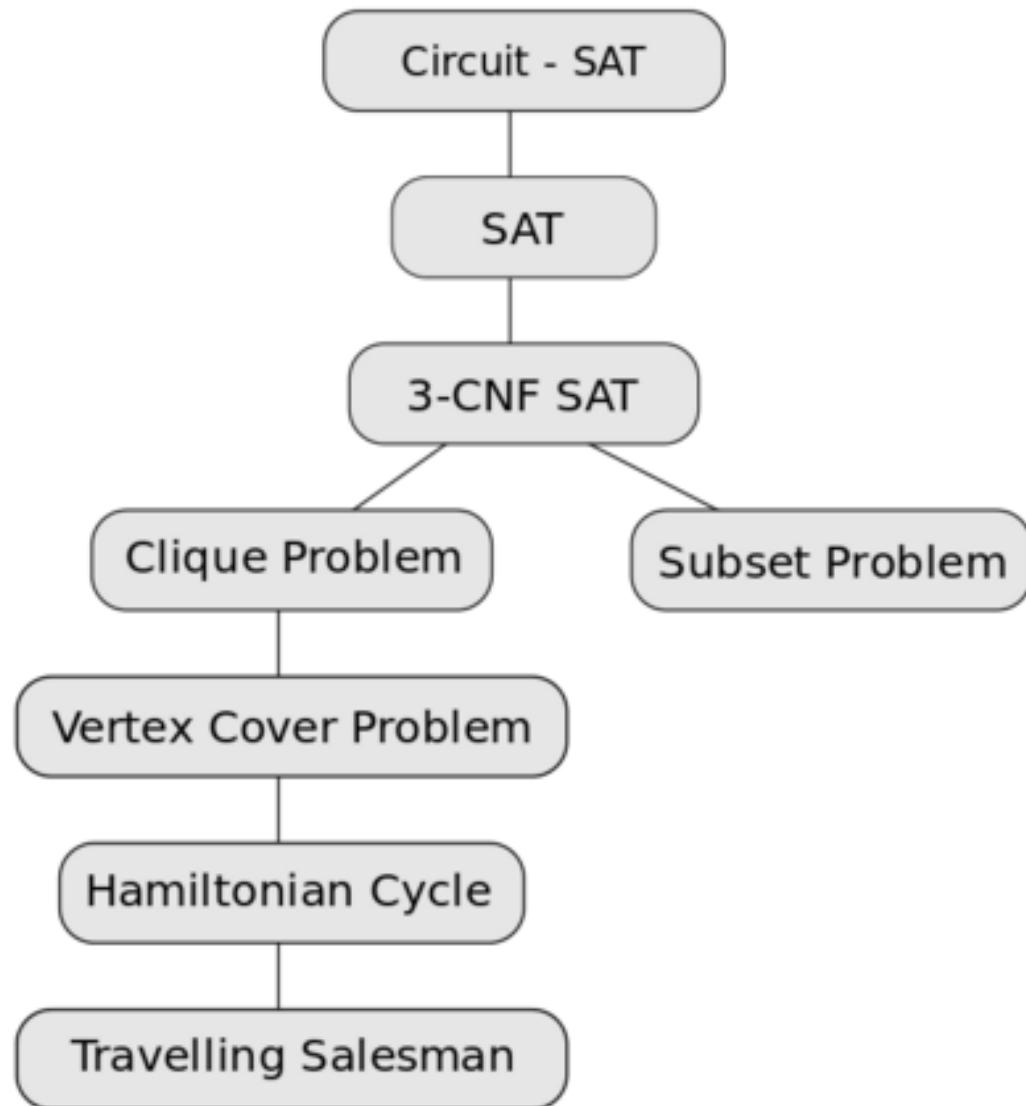
NP Completeness Proofs

- The NP-completeness of a particular problem P relies on the proof that $L \leq_p P$ for every language $L \in NP$.
- Without reducing every language L to P we can do it easily by using the following lemma.
- Lemma:
If L is a language such that $L' \leq_p L$ for some $L' \in NPC$ then L is NP-Hard. Moreover if $L \in NP$ then $L \in NPC$.

Steps to prove a language L is NP-Complete

1. Prove $L \in NP$
2. Select a known NP-Complete language L' .
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ of L .
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0, 1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

NP-Complete Problems



Formula Satisfiability Problem / SAT

An instance of SAT is a Boolean formula ϕ composed of:

1. n Boolean variables x_1, x_2, \dots, x_n
 2. m Boolean connectives: any Boolean function with one or two inputs and one output, such as $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ etc.
 3. Parentheses: atmost one pair of parentheses per Boolean connective.
- Length of the Boolean formula = $n + m$, which is polynomial
 - A formula ϕ with a satisfying assignment is a truth assignment that causes it to evaluate to 1 is a satisfiable formula.
 - Formal Language Definition:
 - $SAT = \{<\phi>: \phi \text{ is a satisfiable boolean formula}\}$
 - Example:
 - The formula $\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ has satisfying assignment of $<x_1=0, x_2=0, x_3=1, x_4=1>$
 - So the formula belongs to SAT.

3-CNF Satisfiability (3 CNF SAT)

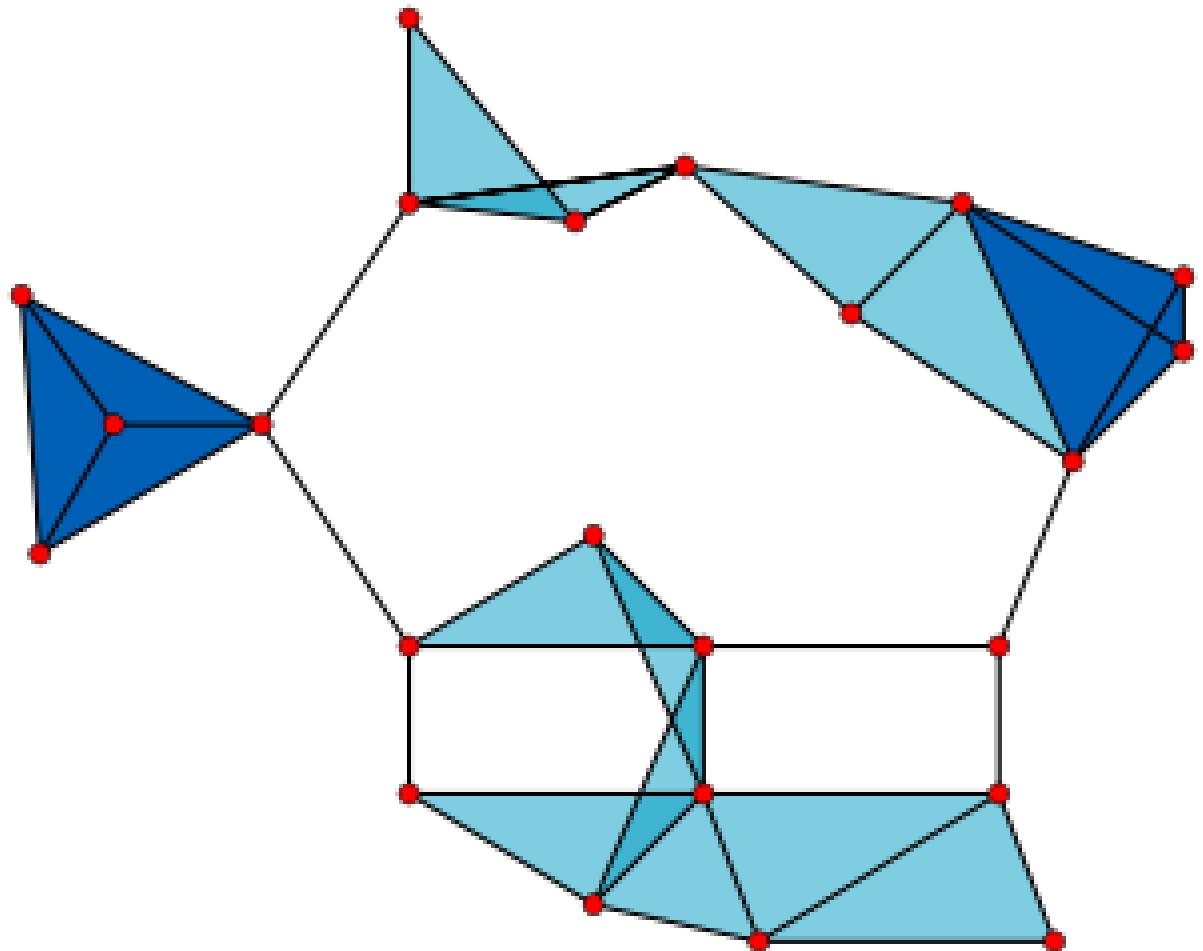
- Literal: A literal in a Boolean formula is an occurrence of a variable or its negation.
- CNF: A Boolean formula is in conjunctive normal form or CNF if it is expressed as an AND of clauses each of which is OR of one or more literals.
- 3CNF: A Boolean formula is in 3CNF if each clause has exactly three distinct literals.
- Example: $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$
- 3 CNF SAT is satisfiable if there exists a truth assignment such that the formula evaluates to TRUE.

CLIQUE

- A **clique**, C , in an undirected graph $G = (V, E)$ is a subset of the vertices, $C \subseteq V$, such that every two distinct vertices are adjacent. This is equivalent to the condition that the subgraph of G induced by C is complete. In some cases, the term clique may also refer to the subgraph directly.
- Size of the clique is no of vertices it contains.
- Formal Language Definition :

CLIQUE = { $\langle G, k \rangle$: G is a graph with a clique of size k }

Clique: Example



A graph with:

- 23×1 -vertex cliques (the vertices),
- 42×2 -vertex cliques (the edges),
- 19×3 -vertex cliques (light and dark blue triangles), and
- 2×4 -vertex cliques (dark blue areas).

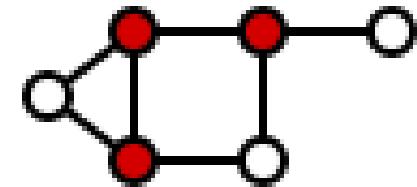
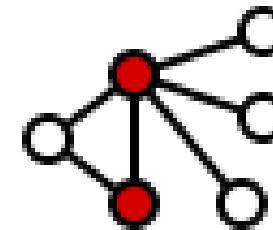
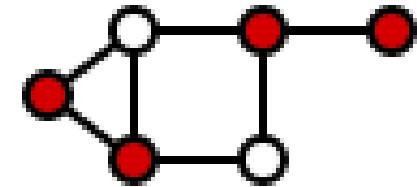
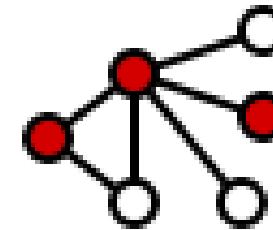
The 11 light blue triangles form maximal cliques. The two dark blue 4-cliques are both maximum and maximal, and the clique number of the graph is 4.

Subset-sum

- Given a finite set $S \subset \mathbb{N}$ and target $t \in \mathbb{N}$. We have to check or find a subset $S' \subseteq S$ whose element sum to t .
- Formal Language Definition :
 $\text{SUBSET-SUM} = \{ \langle S, t \rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{i \in S'} i \}$
- Example: $S = \{ 3, 2, 7, 1 \}$, $t = 6$
Output: TRUE, subset is $\{ 3, 2, 1 \}$

Vertex Cover

- A vertex cover of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ or both.
- Size of a vertex cover is the number of vertices in it.
- Formal Language Definition:
VERTEX-COVER =
 $\{\langle G, k \rangle : \text{graph } G \text{ has a vertex cover of size } k\}$



Hamiltonian Cycle

- A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V .
- A graph that contains a Hamiltonian cycle is called Hamiltonian, otherwise it is nonhamiltonian.
- Formal Language Definition:
 - $\text{HAM-CYCLE} = \{ \langle G \rangle : G \text{ is a Hamiltonian graph} \}$

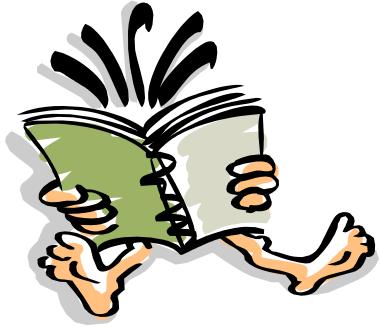
Travelling-Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- It is a problem related to Hamiltonian cycle problem.
- Formal Language Definition:
 $TSP = \{<G, c, k>: G = (V, E) \text{ is a complete graph, } c \text{ is a function from } V \times V \rightarrow \mathbb{Z}, k \in \mathbb{Z} \text{ and } G \text{ has a travelling salesman tour with cost at most } k\}$



Beyond NP: PSPACE

- PSPACE: The class of problems that can be solved using only polynomial amount of space.
 - It is OK to take exponential (or super-exponential) time.
- Key point: Unlike time, space is reusable.
 - Result: many exponential algorithms are in PSPACE.
 - Consider universal formulas. We can check them in polynomial space by rerunning the same computation (say, $\text{check}(v)$) for each v .
 - The space used for check is recycled, but the time adds up for different v 's.
- Note: SAT is in PSPACE
 - Try every possible truth assignment for variables.
 - Thus, all NP-complete problems are in PSPACE.

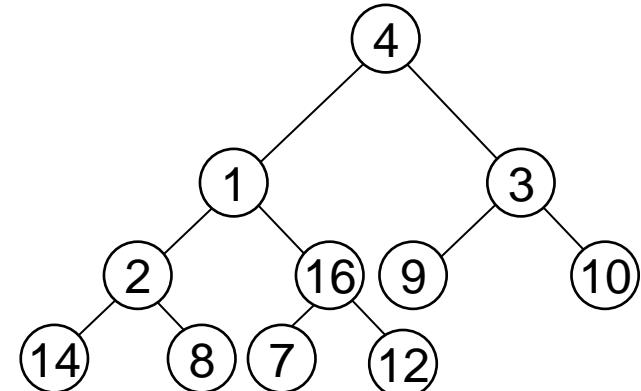


Heapsort

Design and Analysis of Algorithms

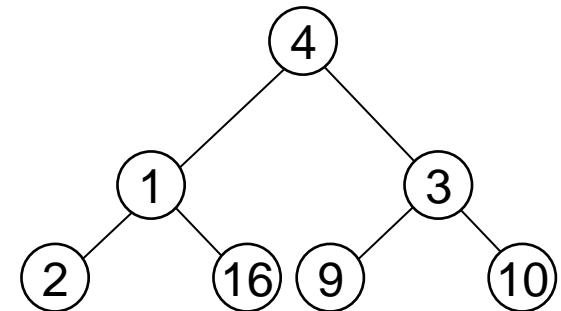
Special Types of Trees

- *Def:* Full binary tree = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

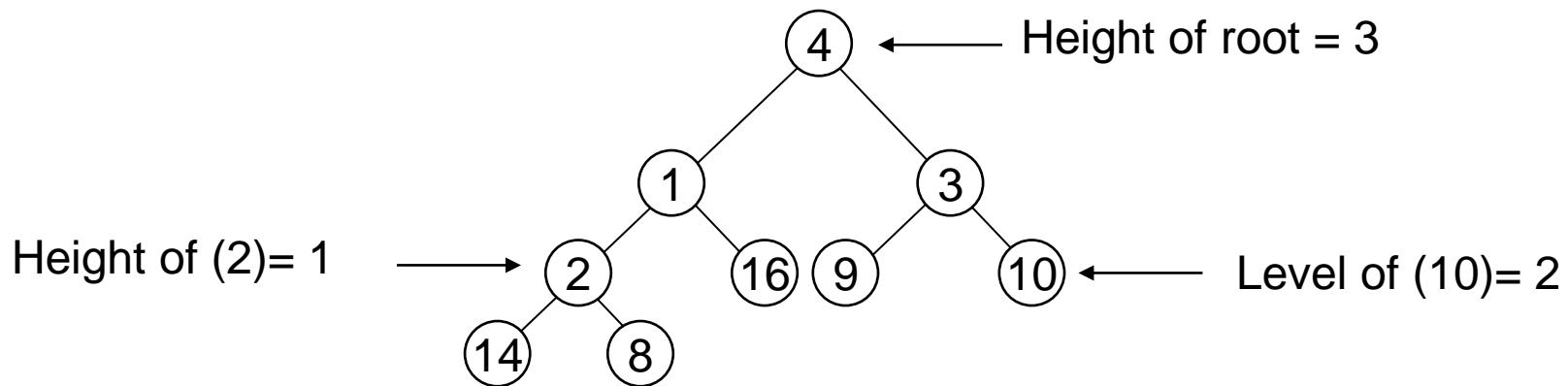
- *Def:* Complete binary tree = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

Definitions

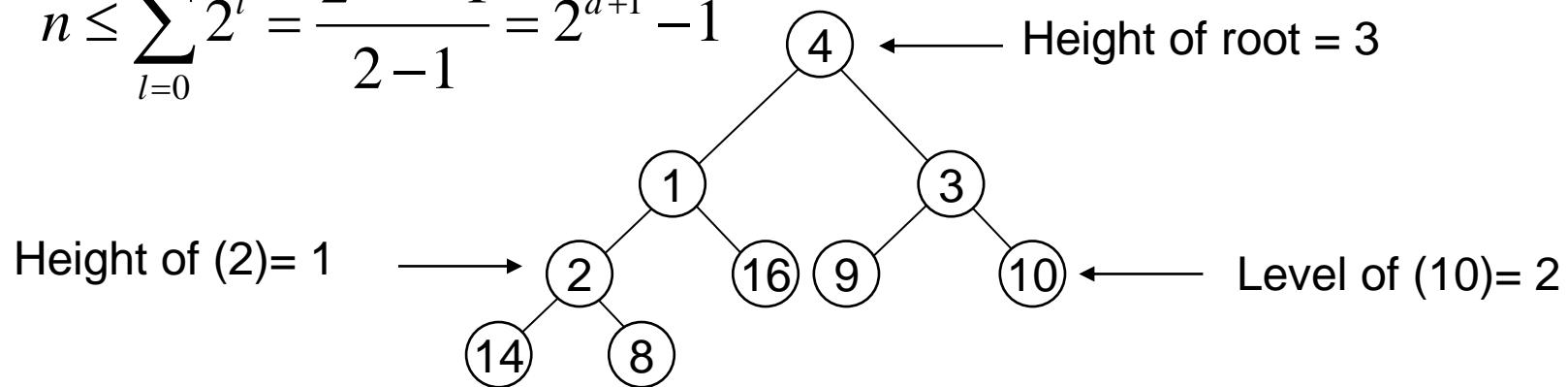
- **Height of a node** = the number of edges on the longest simple path from the node down to a leaf
- **Level of a node** = the length of a path from the root to the node
- **Height of tree** = height of root node



Useful Properties

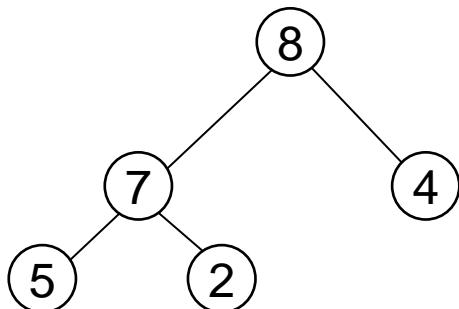
- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with **height** d has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has **height at least** $\lfloor \lg n \rfloor$
(see Ex 6.1-2, page 129)

$$n \leq \sum_{l=0}^d 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$



The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
 - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
 - **Order (heap) property:** for any node x
 $\text{Parent}(x) \geq x$



Heap

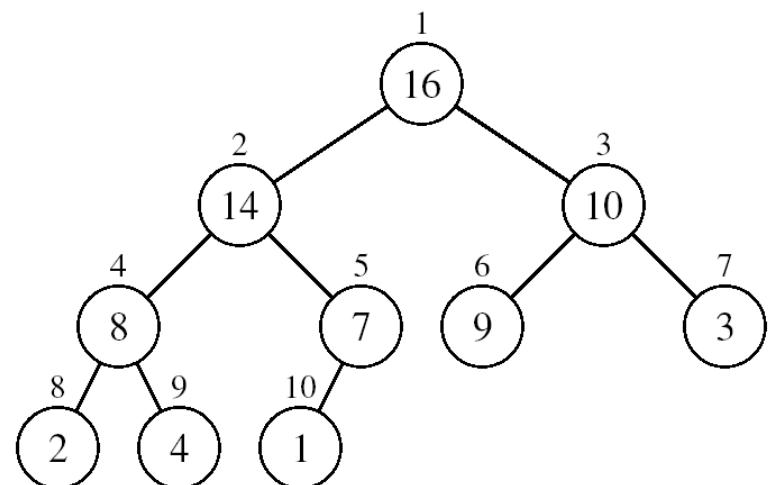
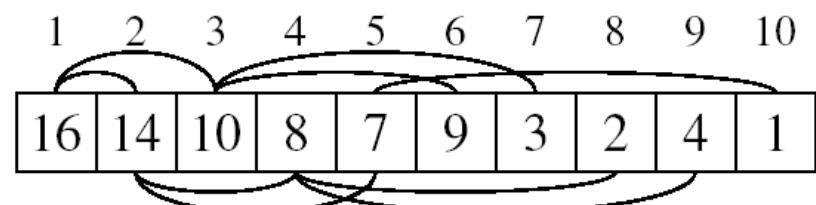
From the heap property, it follows that:

“The root is the maximum element of the heap!”

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array A .
 - Root of tree is $A[1]$
 - Left child of $A[i] = A[2i]$
 - Right child of $A[i] = A[2i + 1]$
 - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
 - Heapsize[A] \leq length[A]
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1 .. n]$ are leaves



Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes i , excluding the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

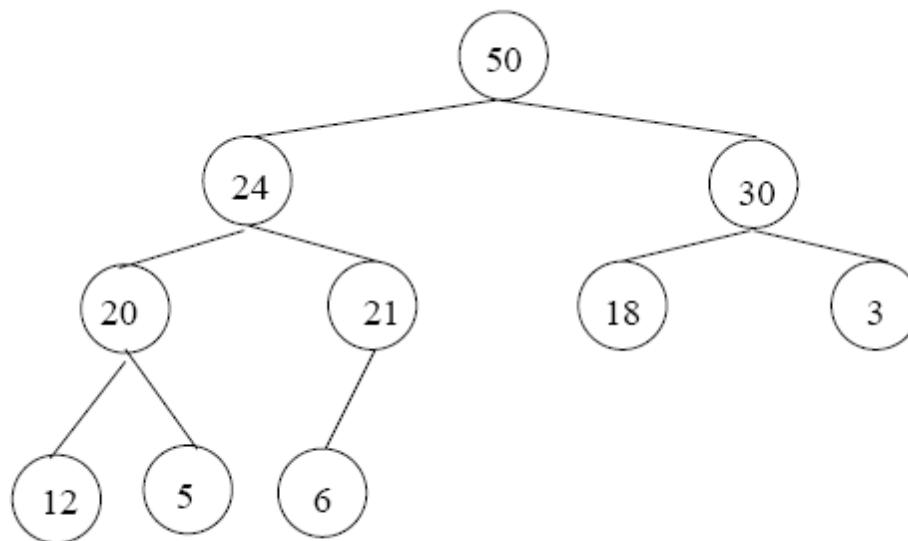
- **Min-heaps** (smallest element at root), have the *min-heap property*:

- for all nodes i , excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

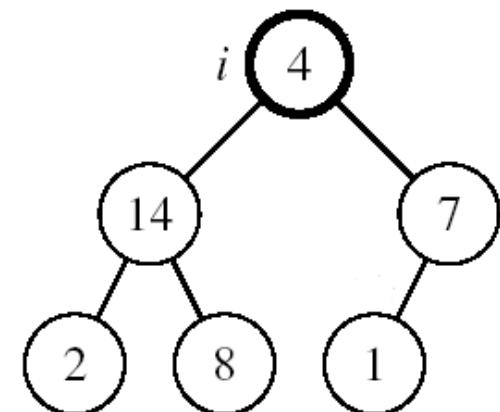


Operations on Heaps

- Maintain/Restore the max-heap property
 - MAX-HEAPIFY
- Create a max-heap from an unordered array
 - BUILD-MAX-HEAP
- Sort an array in place
 - HEAPSORT
- Priority queues

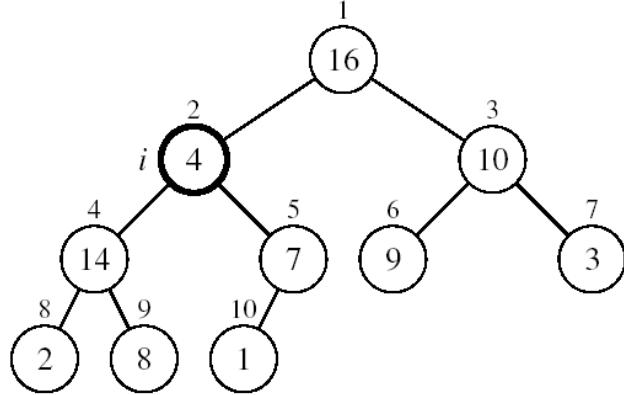
Maintaining the Heap Property

- Suppose a node is smaller than a child
 - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
 - Exchange with larger child
 - Move down the tree
 - Continue until node is not smaller than children

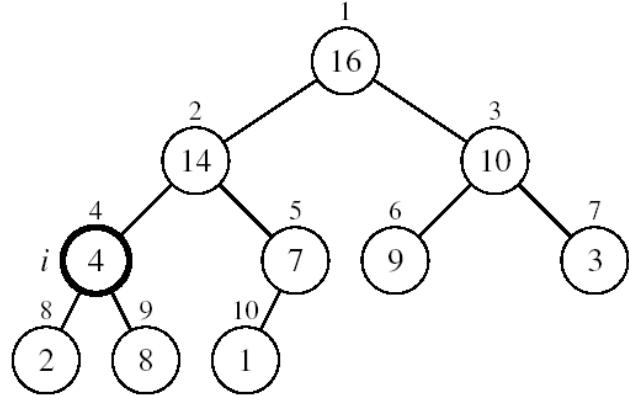


Example

MAX-HEAPIFY(A, 2)



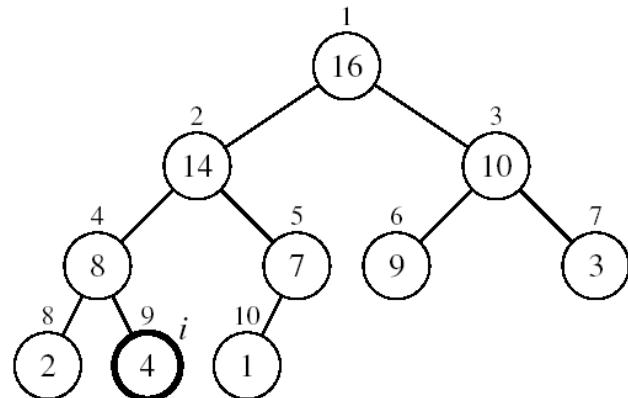
A[2] ↔ A[4]



A[2] violates the heap property

A[4] violates the heap property

A[4] ↔ A[9]



Heap property restored

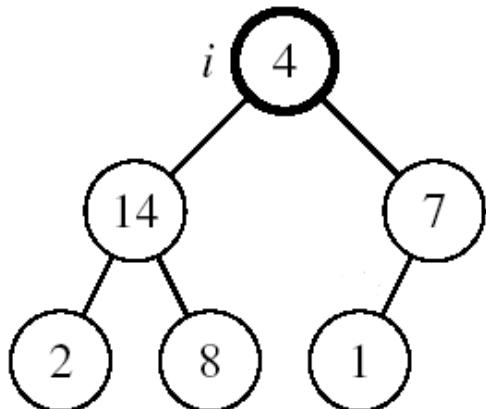
Maintaining the Heap Property

Assumptions:

- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children

Alg: MAX-HEAPIFY(A, i)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq \text{Heap-Size}(A)$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq \text{Heap-Size}(A)$ and $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)



MAX-HEAPIFY Running Time

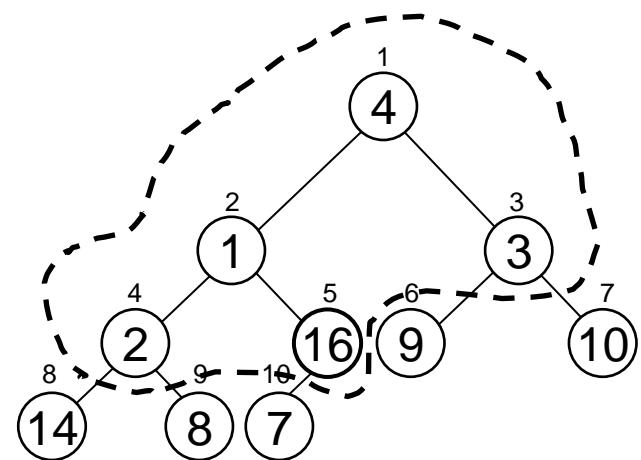
- Intuitively:
 - It traces a path from the root to a leaf.
 - At each level it makes exactly 2 comparisons.
 - Total number of comparisons is $2h$.
 - Running time is $O(h)$ or $O(\lg n)$.
- Running time of MAX-HEAPIFY is $O(\lg n)$
- Can be written in terms of the height of the heap,
as being $O(h)$
 - Since the height of the heap is $\lfloor \lg n \rfloor$

Building a Heap

- Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. $\text{Heap-Size}(A) = \text{length}[A]$
2. **for** $i \leftarrow \lfloor \text{length}[A] / 2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i)



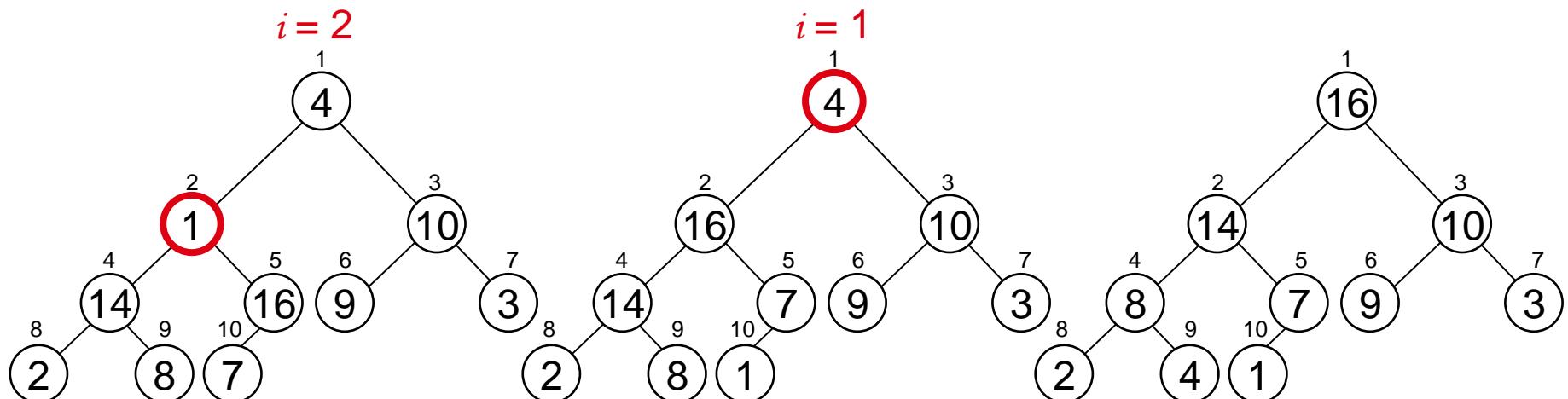
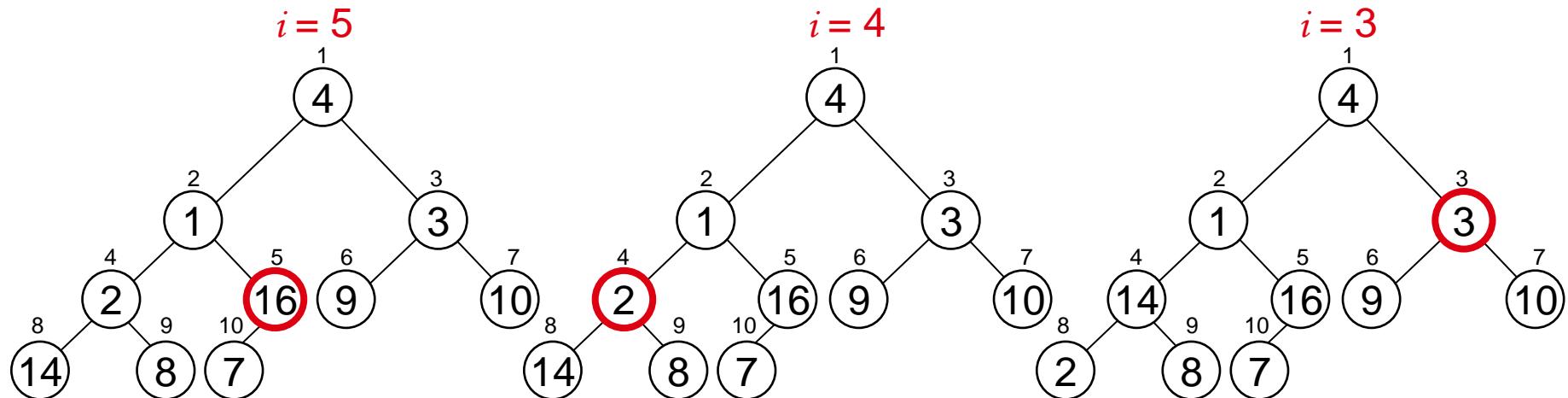
A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

Example:

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. $\text{Heap-Size}(A) = \text{length}[A]$
 2. $\text{for } i \leftarrow \lfloor \text{length}[A] / 2 \rfloor \text{ downto } 1$
 3. $\text{do } \text{MAX-HEAPIFY}(A, i, n)$
- $O(\lg n)$
- $O(n)$

\Rightarrow Running time: $O(n\lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- build-max-heap algorithm executes bottom-to-top.
- Let the size of heap = n
- Max^m no. of elements with height h = $\lceil n/2^{h+1} \rceil$
- When max-heapify is called to a node having height h, the cost is O(h)
- For all nodes of height h, the total cost = $\lceil n/2^{h+1} \rceil * O(h)$

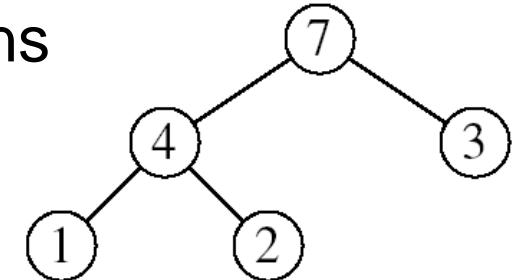
Running Time of BUILD MAX HEAP

For all nodes with varying height, the time complexity =

$$\begin{aligned} & \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil * O(h) \\ &= O\left(n * \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \\ &\leq O\left(n * \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O\left(n * \frac{1/2}{(1 - 1/2)^2}\right) \\ \Rightarrow T(n) &= O(n) \end{aligned}$$

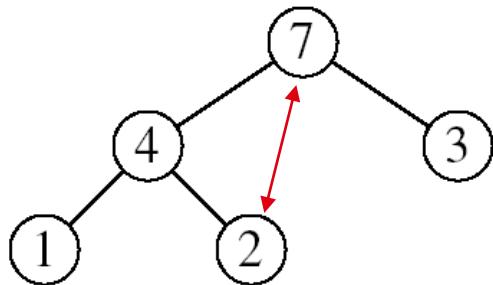
Heapsort

- Goal:
 - Sort an array using heap representations
- Idea:
 - Build a **max-heap** from the array
 - Swap the root (the maximum element) with the last element in the array
 - “Discard” this last node by decreasing the heap size
 - Call **MAX-HEAPIFY** on the new root
 - Repeat this process until only one node remains

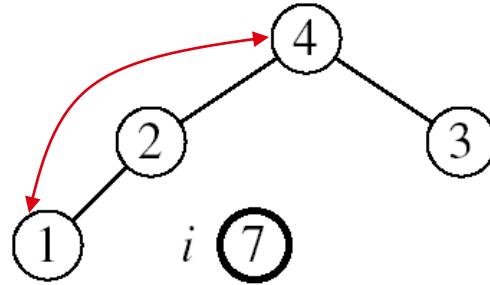


Example:

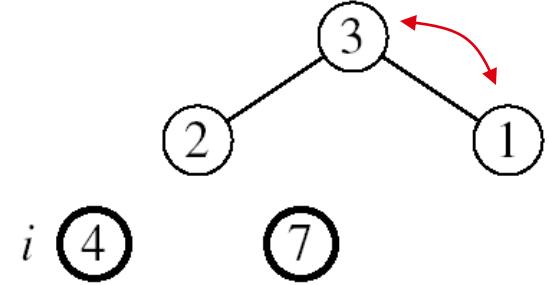
$A=[7, 4, 3, 1, 2]$



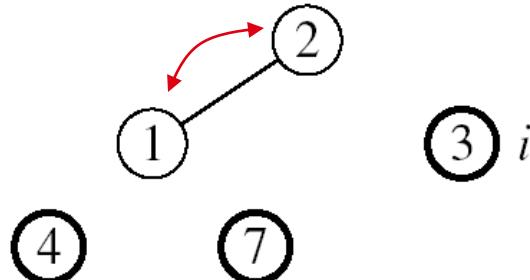
EXCHANGE $A[1] \leftrightarrow A[5]$
MAX-HEAPIFY(A , 1)



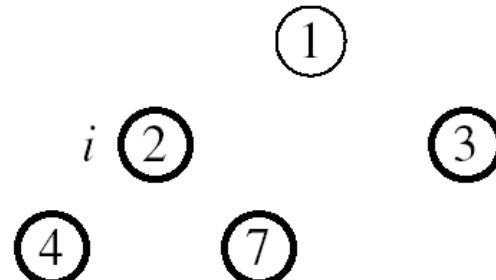
EXCHANGE $A[1] \leftrightarrow A[4]$
MAX-HEAPIFY(A , 1)



EXCHANGE $A[1] \leftrightarrow A[3]$
MAX-HEAPIFY(A , 1)



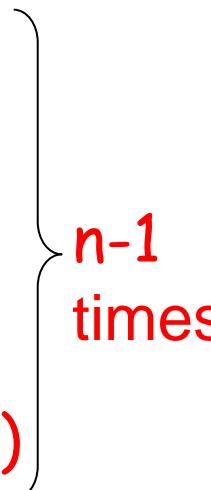
EXCHANGE $A[1] \leftrightarrow A[2]$
MAX-HEAPIFY(A , 1)



A

1	2	3	4	7
---	---	---	---	---

Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A) $O(n)$
 2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
 3. **do** exchange $A[1] \leftrightarrow A[i]$ $\theta(1)$
 4. Heap-Size(A) = Heap-Size(A) - 1 $\theta(1)$
 5. MAX-HEAPIFY($A, 1$) $O(\lg n)$
- Running time: $O(n \lg n)$
- 

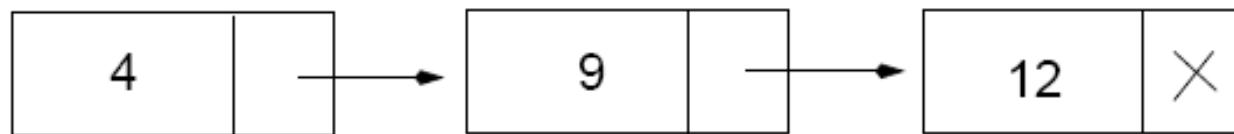
Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first
- Major operations

Remove an element from the queue

Insert an element in the queue



Operations on Priority Queues

- Max-priority queues support the following operations:
 - $\text{INSERT}(S, x)$: inserts element x into set S
 - $\text{EXTRACT-MAX}(S)$: removes and returns element of S with largest key
 - $\text{MAXIMUM}(S)$: returns element of S with largest key
 - $\text{INCREASE-KEY}(S, x, k)$: increases value of element x 's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

Goal:

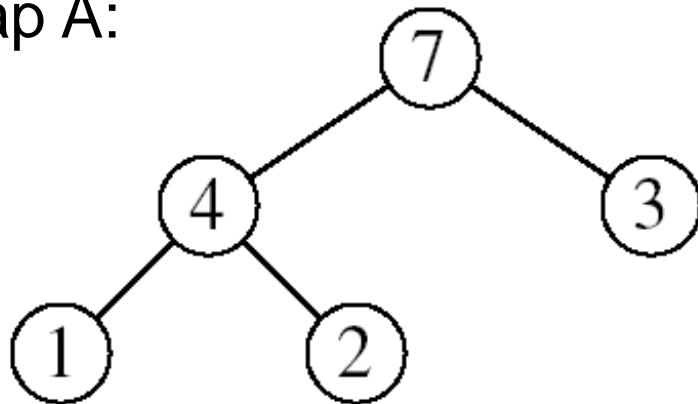
- Return the largest element of the heap

Alg: HEAP-MAXIMUM(A)

Running time: $O(1)$

1. **return $A[1]$**

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

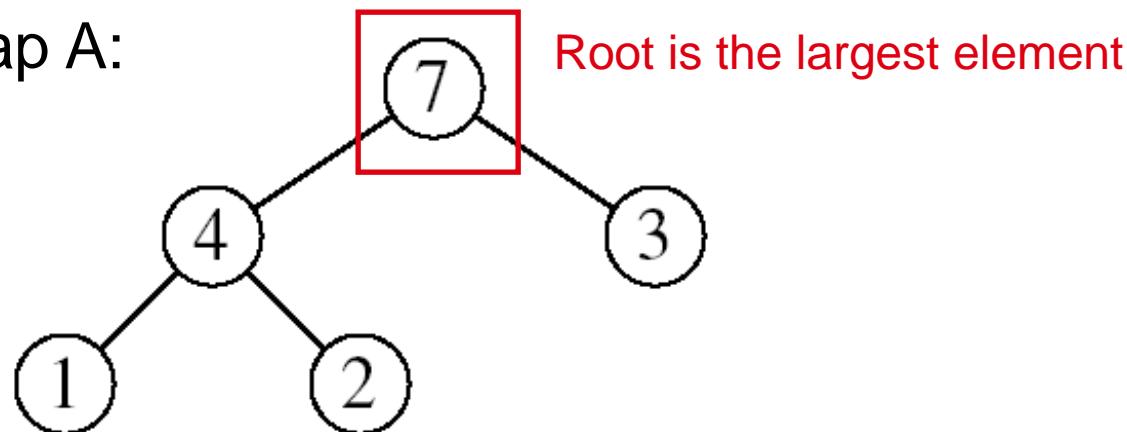
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap)

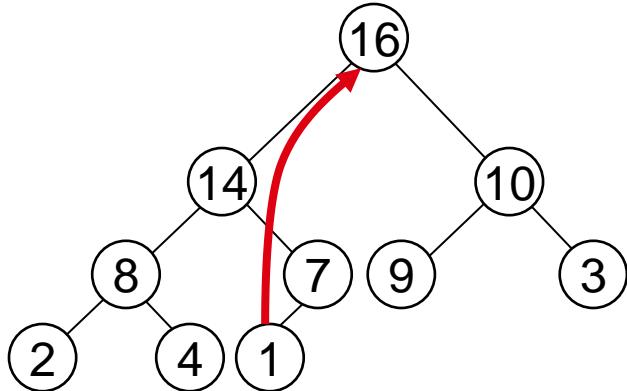
Idea:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size $n-1$

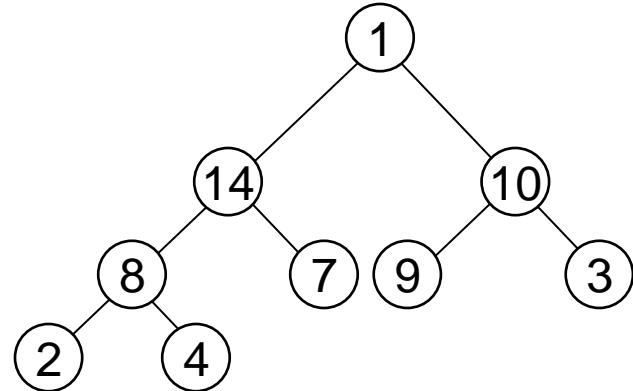
Heap A:



Example: HEAP-EXTRACT-MAX

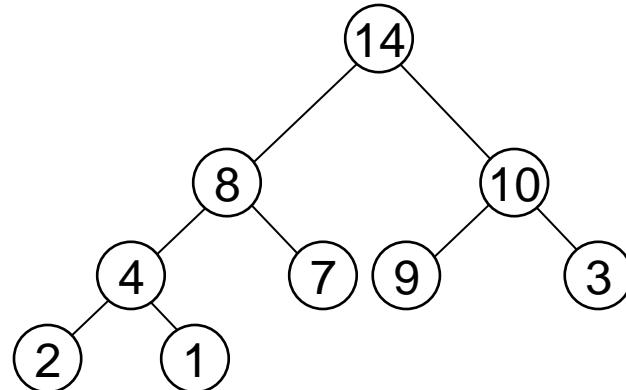


max = 16



Heap size decreased with 1

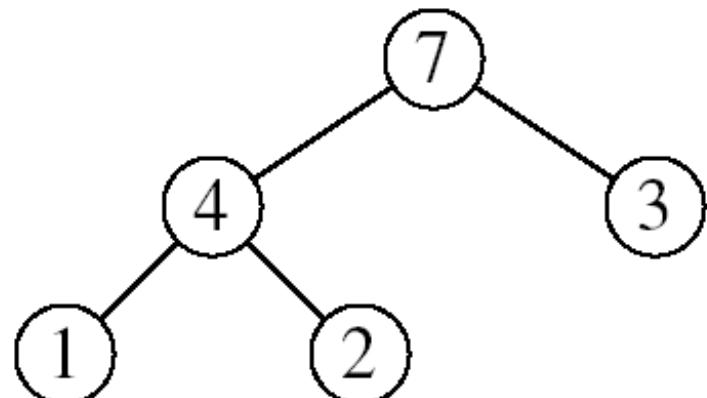
Call MAX-HEAPIFY($A, 1$)



HEAP-EXTRACT-MAX

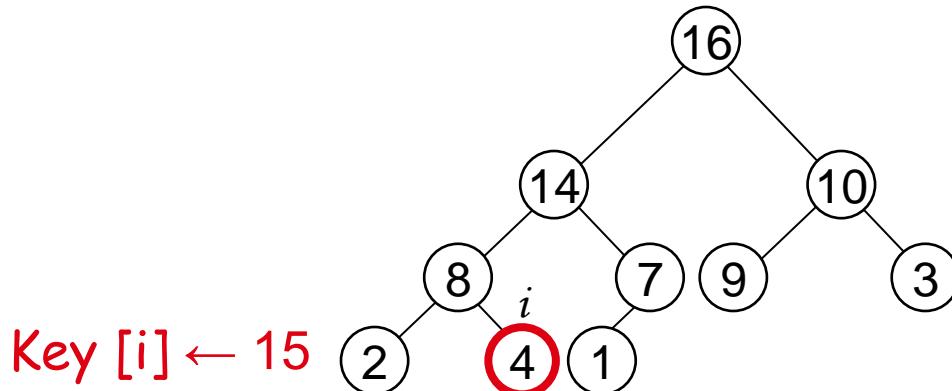
Alg: HEAP-EXTRACT-MAX(A)

1. **if** $\text{heap-size}[A] < 1$
2. **then error** “heap underflow”
3. $\text{max} \leftarrow A[1]$
4. $A[1] \leftarrow A[\text{heap-size}[A]]$
5. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
6. MAX-HEAPIFY($A, 1$) ▷ remakes heap
7. **return** max

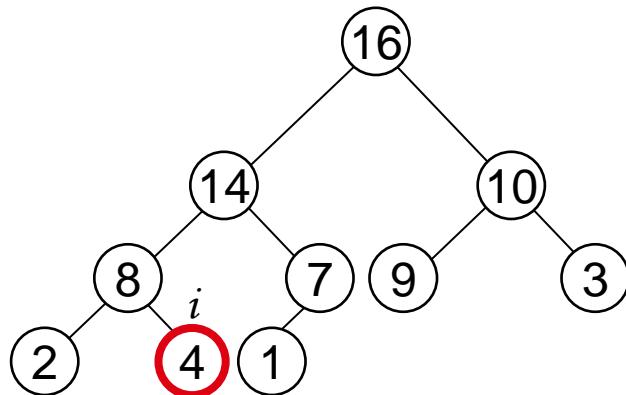


HEAP-INCREASE-KEY

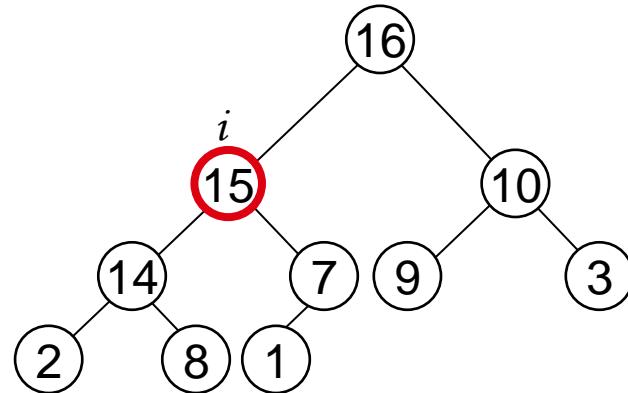
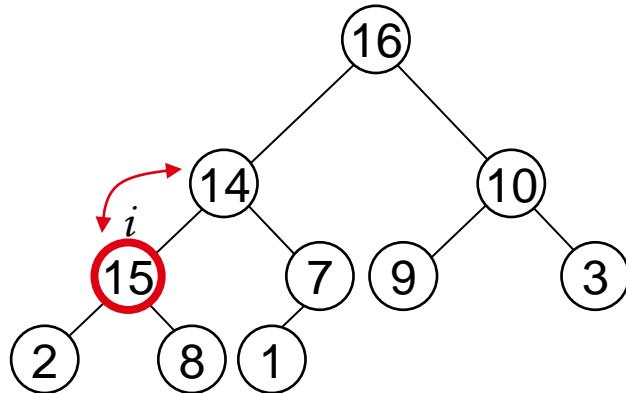
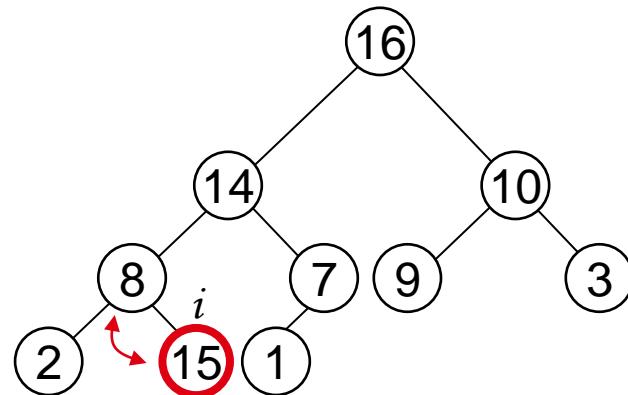
- Goal:
 - Increases the key of an element i in the heap
- Idea:
 - Increment the key of $A[i]$ to its new value
 - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



Example: HEAP-INCREASE-KEY



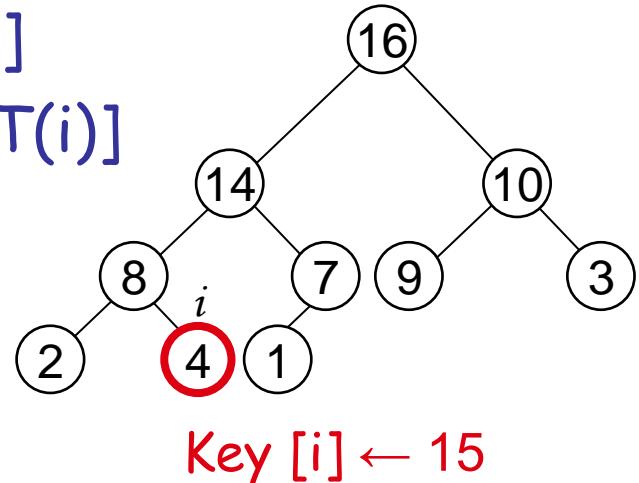
$\text{Key}[i] \leftarrow 15$



HEAP-INCREASE-KEY

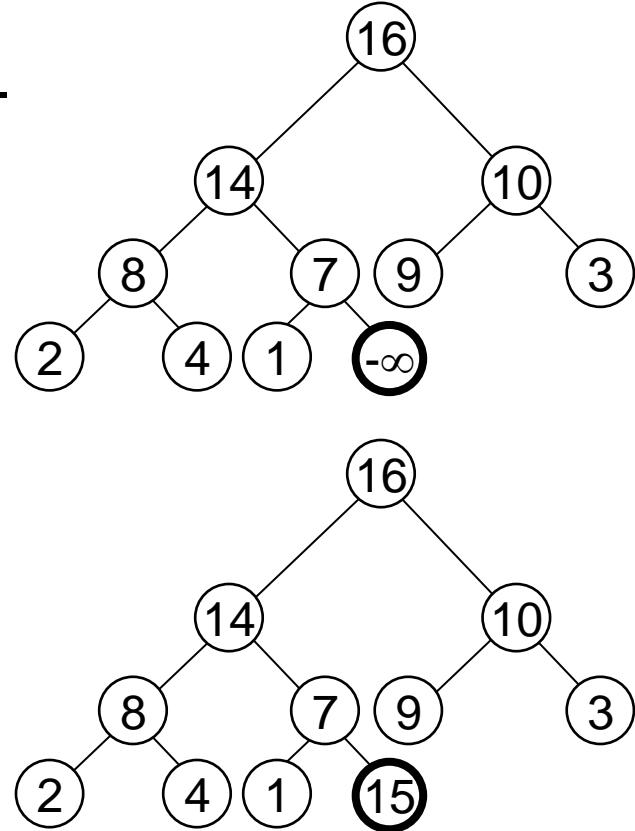
Alg: HEAP-INCREASE-KEY(A , i , key)

1. **if** key < $A[i]$
 2. **then error** “new key is smaller than current key”
 3. $A[i] \leftarrow \text{key}$
 4. **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
 5. **do exchange** $A[i] \leftrightarrow A[\text{PARENT}(i)]$
 6. $i \leftarrow \text{PARENT}(i)$
- Running time: $O(\lg n)$



MAX-HEAP-INSERT

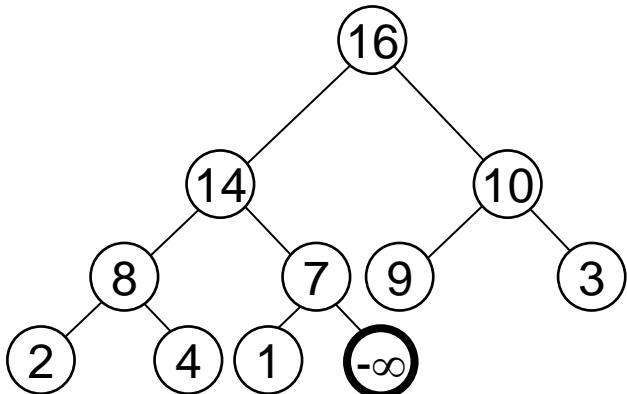
- Goal:
 - Inserts a new element into a max-heap
- Idea:
 - Expand the max-heap with a new element whose key is $-\infty$
 - Calls **HEAP-INCREASE-KEY** to set the key of the new node to its correct value and maintain the max-heap property



Example: MAX-HEAP-INSERT

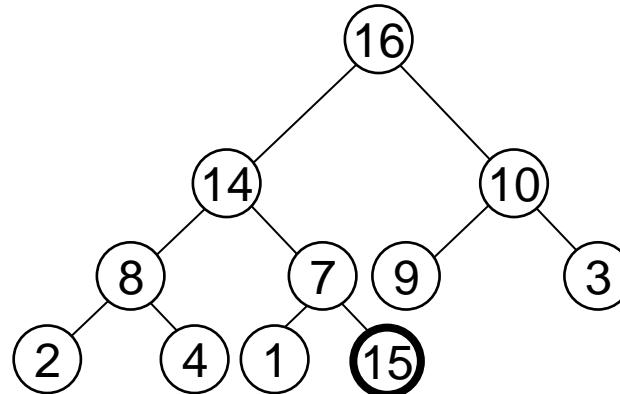
Insert value 15:

- Start by inserting $-\infty$

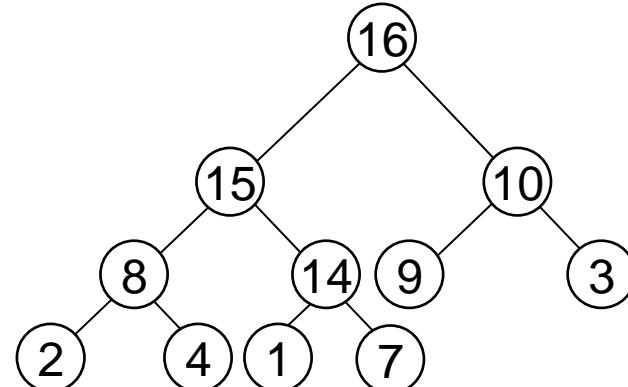
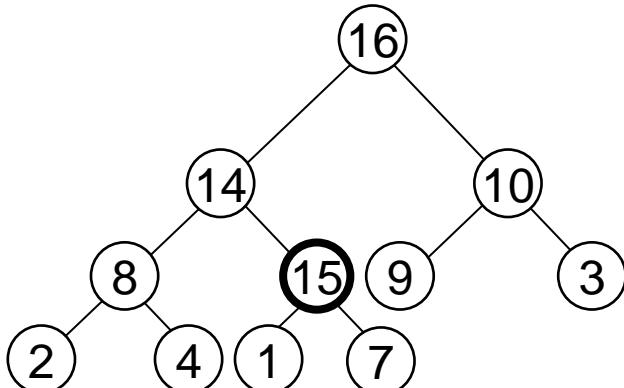


Increase the key to 15

Call HEAP-INCREASE-KEY on $A[11] = 15$



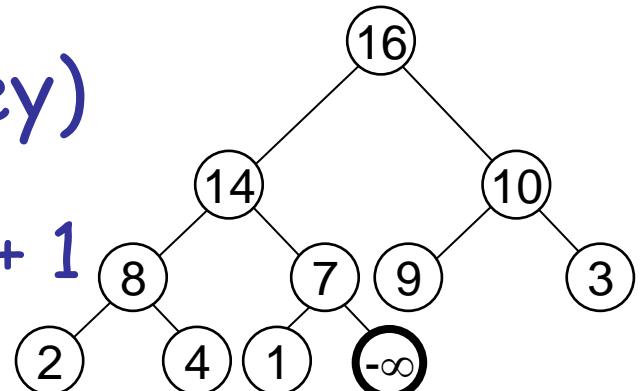
The restored heap containing
the newly added element



MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(A , key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $A[\text{heap-size}[A]] \leftarrow -\infty$
3. HEAP-INCREASE-KEY(A , $\text{heap-size}[A]$, key)



Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:

– MAX-HEAPIFY	$O(lgn)$
– BUILD-MAX-HEAP	$O(n)$
– HEAP-SORT	$O(nlgn)$
– MAX-HEAP-INSERT	$O(lgn)$
– HEAP-EXTRACT-MAX	$O(lgn)$
– HEAP-INCREASE-KEY	$O(lgn)$
– HEAP-MAXIMUM	$O(1)$

Average $O(lgn)$

Knuth-Morris-Pratt (KMP) Algorithm

Ajit Kumar behera

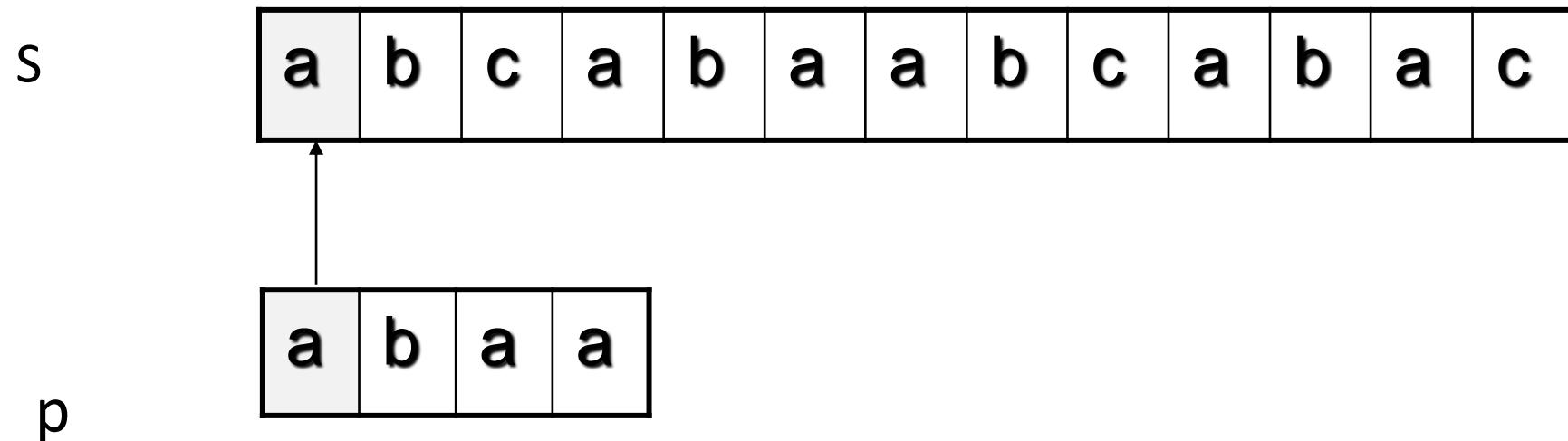
The problem of String Matching

- Given a string S , the problem of string matching deals with finding whether a pattern p occurs in S and
- If p does occur then returning position in S where p occurs.

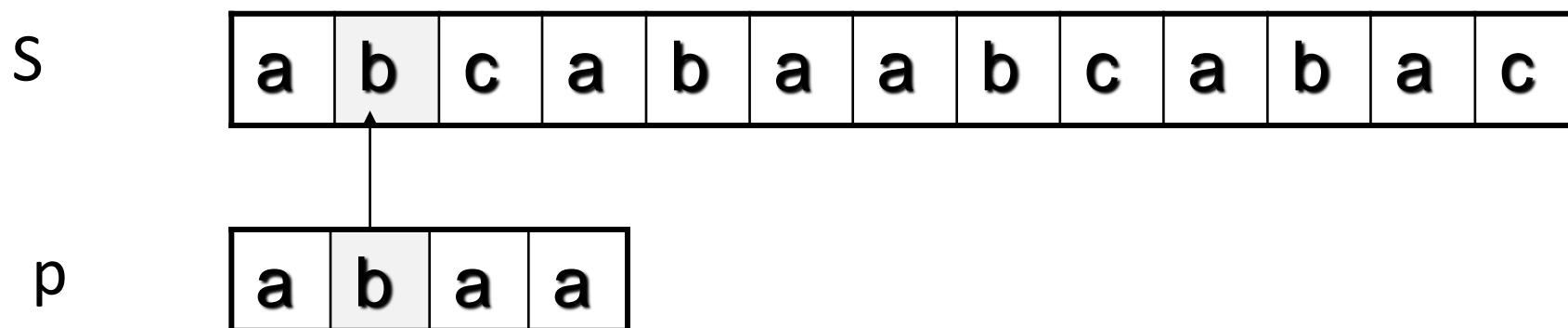
.... a O($m n$) approach

- Compare the first element of the pattern ‘p’ to be searched with the first element of the string ‘S’.
- If the first element of ‘p’ matches the first element of ‘S’, compare the second element of ‘p’ with second element of ‘S’.
- If match found, proceed likewise until entire ‘p’ is found.
- If a mismatch is found at any position, shift ‘p’ one position to the right and repeat comparison beginning from first element of ‘p’.

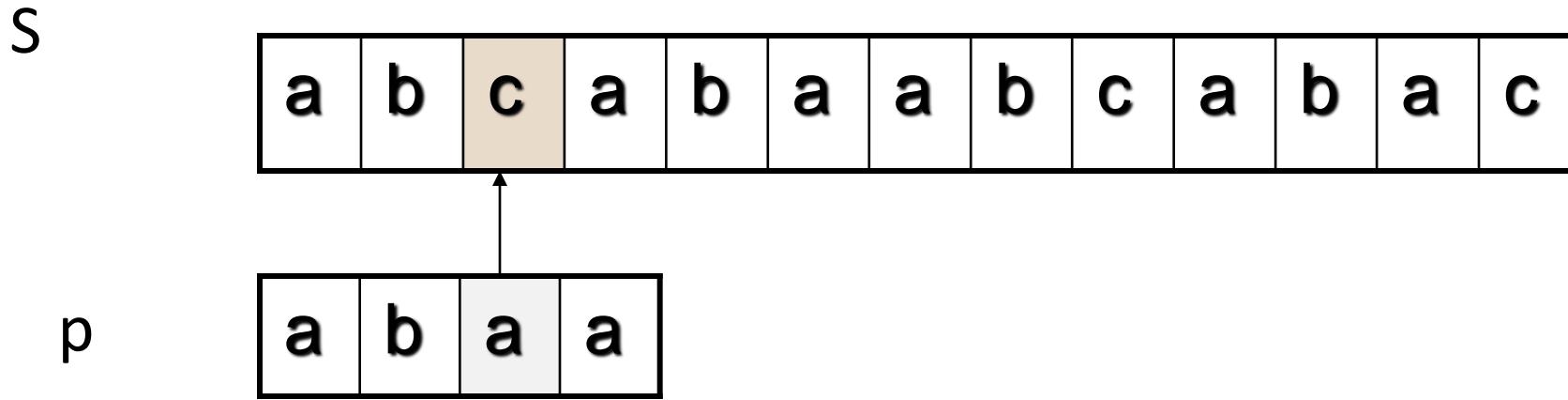
Step 1: compare p[1] with S[1]



Step 2: compare p[2] with S[2]



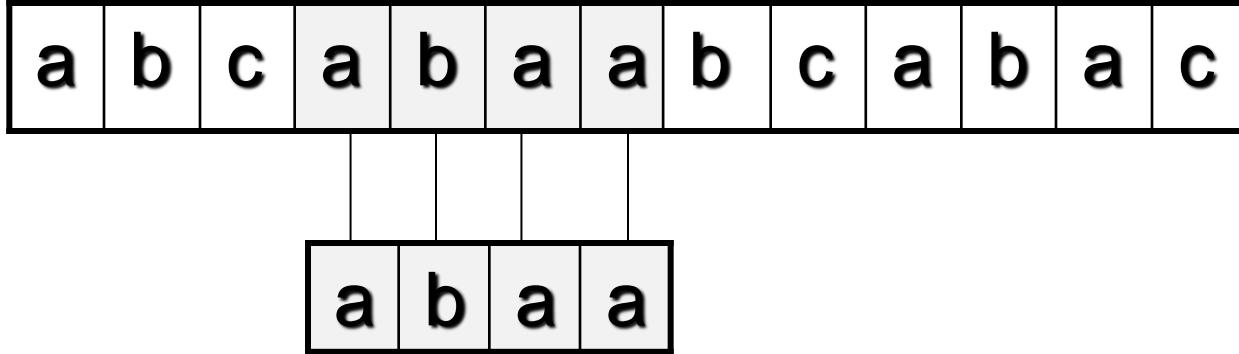
Step 3: compare $p[3]$ with $S[3]$



Mismatch occurs here..

At position where mismatch is detected, shift 'p'
one position to the right and repeat matching procedure.

- S



- Finally, a match would be found after shifting ' p ' three times to the right side.

Drawbacks of this approach: if ' m ' is the length of pattern ' p ' and ' n ' the length of string ' S ', the matching time is of the order $O(mn)$.

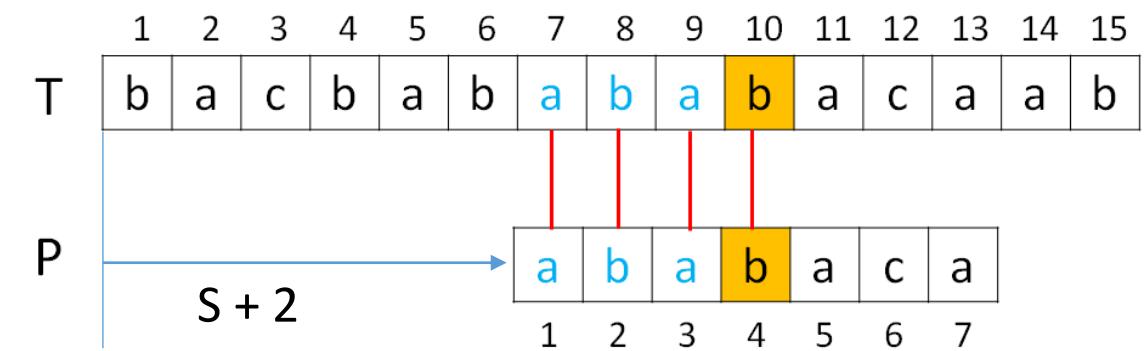
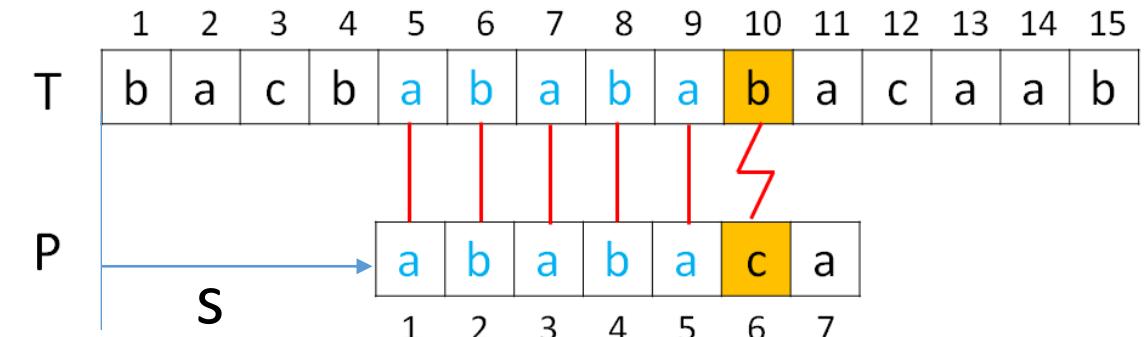
Knuth-Morris-Pratt (KMP) Algorithm

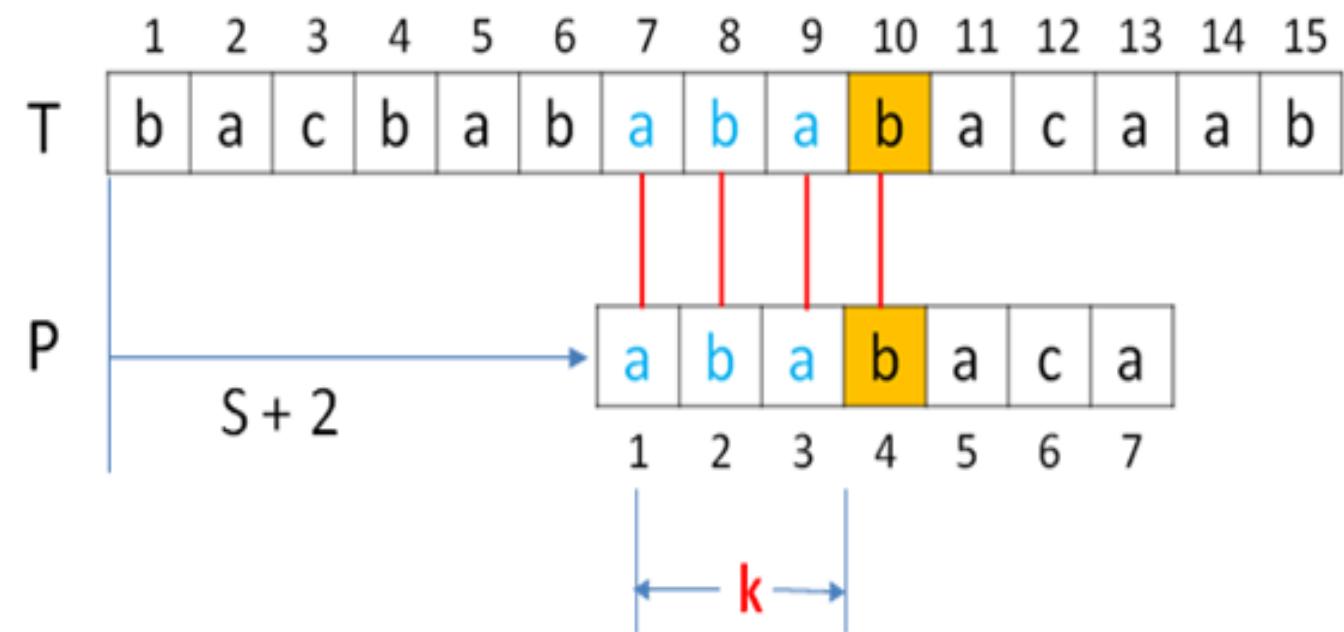
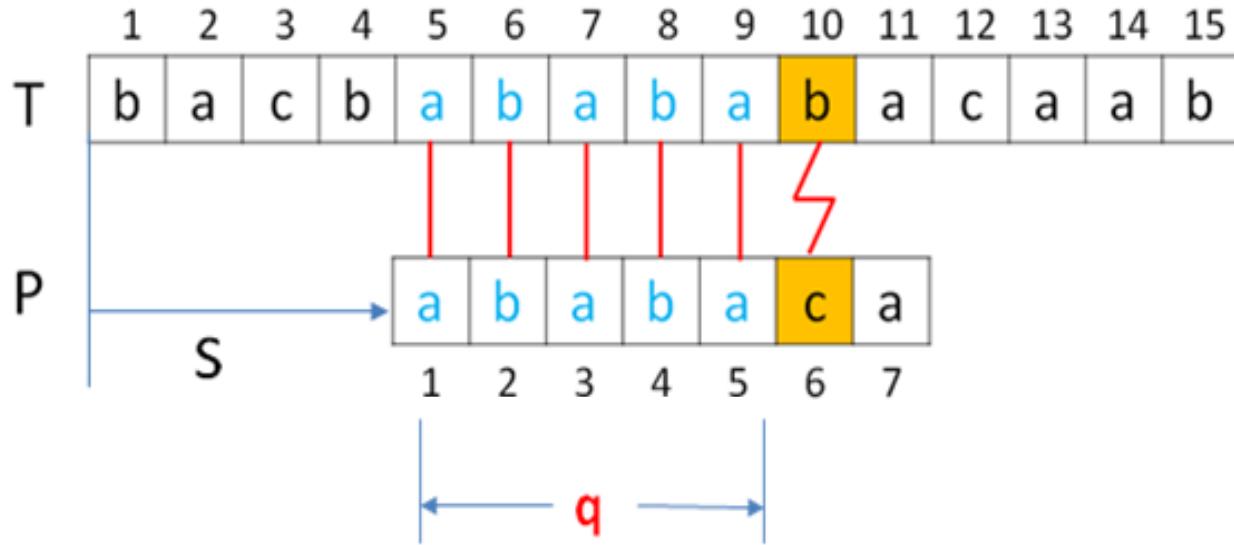
Main Features:

- Best known for linear time for exact matching.
- Compares the pattern to the text from left to right but shifts the pattern intelligently (some time more than one position) than brute force algorithm.
- Pre-processing approach of pattern to avoid trivial comparisons.
- Avoids re-computing matches.

Motivation

- Here $q=5$ characters of the pattern have matched successfully for shift s .
- The $(q+1)^{\text{th}}$ character fails to match the corresponding text character.
- Knowing these q text characters allows us to determine that certain shifts are invalid.
- Here $s+1$ is invalid.
- In shift $s+2$ first three characters need not be matched again.





How to find k?

- To find the value of k we need to precompute the necessary information by comparing the pattern against itself.
- We use an array Π to store this information.
- So the i^{th} entry of Π i.e. $\Pi[i]$ stores the length of longest prefix of $P[1..i]$ that is a suffix of $P[2..i]$.
- The prefix function also indicates how much of the last comparison can be reused if it fails.

Computation of Prefix Function Π

Algorithm *COMPUTE-PREFIX-FUNCTION(P)*

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
6. while $k > 0$ and $P[k+1] \neq P[q]$
7. $k \leftarrow \Pi[k]$
8. if $P[k+1] = P[q]$
9. $k \leftarrow k + 1$
10. $\Pi[q] \leftarrow k$
11. return Π

Computation of Prefix Function: Example

Initially:

$m=7$

$\Pi[1] = 0$

$k = 0$

Π	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>								1	2	3	4	5	6	7
1	2	3	4	5	6	7									

Step 1:

$q = 2$

$k = 0$

$P[1] \neq P[2]$

$\Pi[2] = 0$

Π	<table border="1"><tr><td>0</td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	0	0						1	2	3	4	5	6	7
0	0														
1	2	3	4	5	6	7									

Algorithm COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
 - 6. while $k > 0$ and $P[k+1] \neq P[q]$
 - 7. $k \leftarrow \Pi[k]$
 - 8. if $P[k+1] = P[q]$
 - 9. $k \leftarrow k + 1$
 - 10. $\Pi[q] \leftarrow k$
 - 11. return Π

P	<table border="1"><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>a</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	a	b	a	b	a	c	a	1	2	3	4	5	6	7
a	b	a	b	a	c	a									
1	2	3	4	5	6	7									

Computation of Prefix Function: Example

Initially:

$m=7$

$\Pi[1] = 0$

$k = 0$

Π	0						
	1	2	3	4	5	6	7

Step 1:

$q = 2$

$k = 0$

$P[1] \neq P[2]$

$\Pi[2] = 0$

Π	0	0					
	1	2	3	4	5	6	7

Algorithm COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
 while $k > 0$ and $P[k+1] \neq P[q]$
 $k \leftarrow \Pi[k]$
 if $P[k+1] = P[q]$
 $k \leftarrow k + 1$
 $\Pi[q] \leftarrow k$
11. return Π

P	a	b	a	b	a	c	a
	1	2	3	4	5	6	7

Computation of Prefix Function: Example

Step 2:

$q = 3$

$k = 0$

$P[1] = P[3]$

k becomes 1

$\Pi[3] = 1$

Π	0	0	1				
	1	2	3	4	5	6	7

Step 3:

$q = 4$

$k = 1$

$P[2] = P[4]$

k becomes 2

$\Pi[4] = 2$

Π	0	0	1	2			
	1	2	3	4	5	6	7

Algorithm COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
 while $k > 0$ and $P[k+1] \neq P[q]$
 $k \leftarrow \Pi[k]$
 if $P[k+1] = P[q]$
 $k \leftarrow k + 1$
 $\Pi[q] \leftarrow k$
11. return Π

P	a	b	a	b	a	c	a
	1	2	3	4	5	6	7

Computation of Prefix Function: Example

Step 4:

$q = 5$

$k = 2$

$P[3] = P[5]$

k becomes 3

$\Pi[q] = 3$

Π	0	0	1	2	3		
	1	2	3	4	5	6	7

Step 5:

$q = 6$

$k = 3$

$P[3] \neq P[6]$

$k = \Pi[3] = 1$

$P[2] \neq P[6]$

$k = \Pi[1] = 0$

$\Pi[6] = 0$

Π	0	0	1	2	3	0	
	1	2	3	4	5	6	7

Algorithm COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
 while $k > 0$ and $P[k+1] \neq P[q]$
 $k \leftarrow \Pi[k]$
 if $P[k+1] = P[q]$
 $k \leftarrow k + 1$
 $\Pi[q] \leftarrow k$
11. return Π

P	a	b	a	b	a	c	a
	1	2	3	4	5	6	7

Computation of Prefix Function: Example

Step 6:

$q = 7$

$k = 0$

$P[1] = P[7]$

k becomes 1

$\Pi[7] = 1$

Π	0	0	1	2	3	0	1
	1	2	3	4	5	6	7

Algorithm COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow \text{length}[P]$
2. Let $\Pi[1 \dots m]$ be a new array
3. $\Pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. for $q \leftarrow 2$ to m
 - 6. while $k > 0$ and $P[k+1] \neq P[q]$
 - 7. $k \leftarrow \Pi[k]$
 - 8. if $P[k+1] = P[q]$
 - 9. $k \leftarrow k + 1$
 - 10. $\Pi[q] \leftarrow k$
 - 11. return Π

P	a	b	a	b	a	c	a
	1	2	3	4	5	6	7

KMP Matching Algorithm

Algorithm KMP-MATCHER(T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$ // number of characters matched
5. for $i \leftarrow 1$ to n
6. while $q > 0$ and $P[q+1] \neq T[i]$ // next character does not match
7. $q \leftarrow \Pi[q]$ // look for the next match
8. if $P[q+1] = T[i]$ // next character matches
9. $q \leftarrow q + 1$
10. if $q = m$
11. Print “Pattern occurs with shift” $i - m$
12. $q \leftarrow \Pi[q]$ // look for the next match

Initially:

$n=15$

$m=7$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
P	a	b	a	b	a	c	a								
Π	0	0	1	2	3	0	1								

Step 1:

$i=1$

$q=0$

$P[1]$ does not match with $T[1]$.

P will be shifted one position to the right.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
P	a	b	a	b	a	c	a								

Algorithm KMP-MATCHER(T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$
5. for $i \leftarrow 1$ to n
6. while $q > 0$ and $P[q+1] \neq T[i]$
7. $q \leftarrow \Pi[q]$
8. if $P[q+1] = T[i]$
9. $q \leftarrow q + 1$
10. if $q = m$
11. Print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$

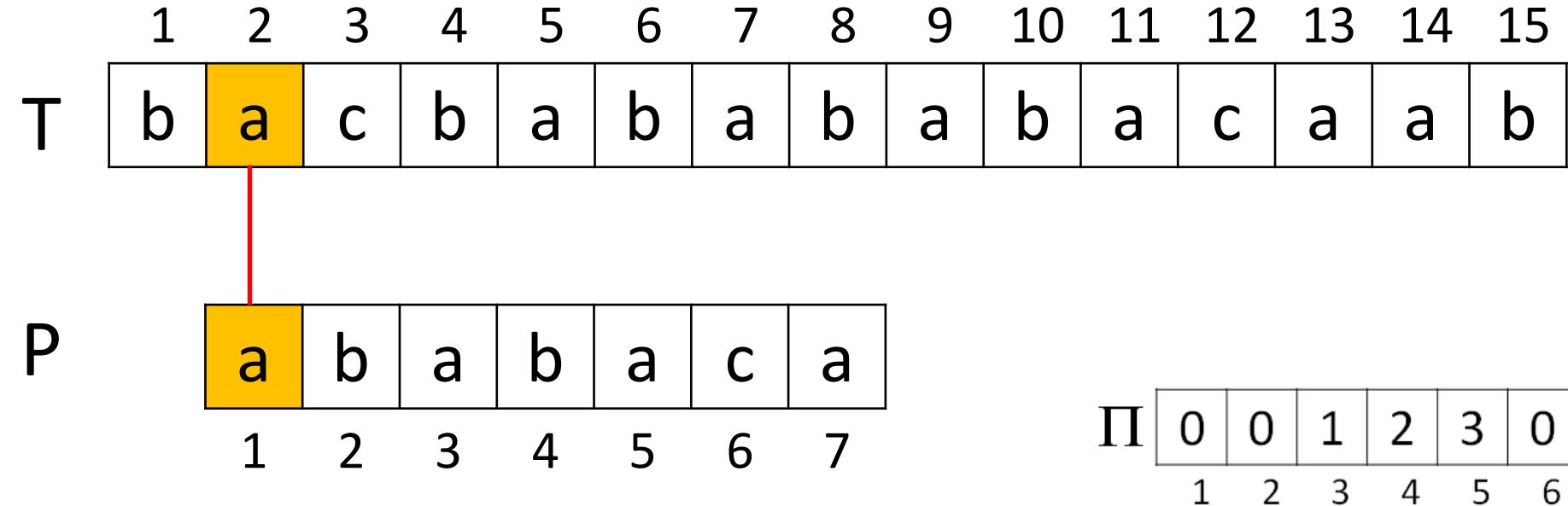
Step 2:

i=2

q=0

P[1] match with T[2].

q is increased to 1.



Step 3:

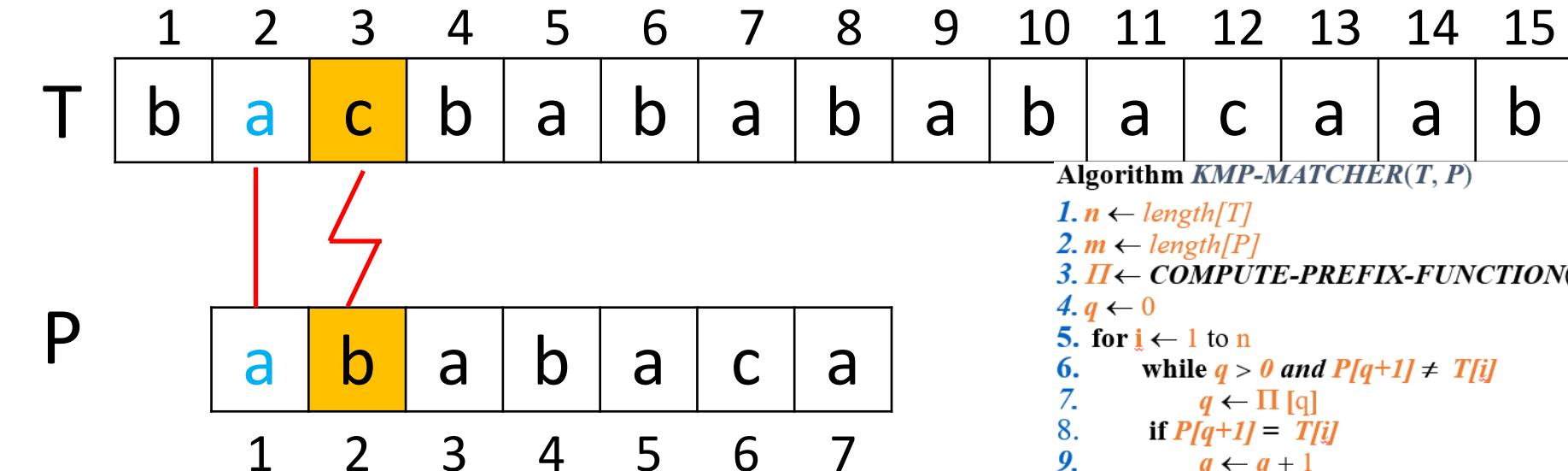
i=3

q=1

P[2] does not
match with T[3].

$q = \Pi[q] = \Pi[1] = 0$

P[1] does not
match with T[3].



Algorithm **KMP-MATCHER(T, P)**

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$
5. **for** $i \leftarrow 1$ to n
6. **while** $q > 0$ and $P[q+1] \neq T[i]$
7. $q \leftarrow \Pi[q]$
8. **if** $P[q+1] = T[i]$
9. $q \leftarrow q + 1$
10. **if** $q = m$
11. Print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$

Step 4:

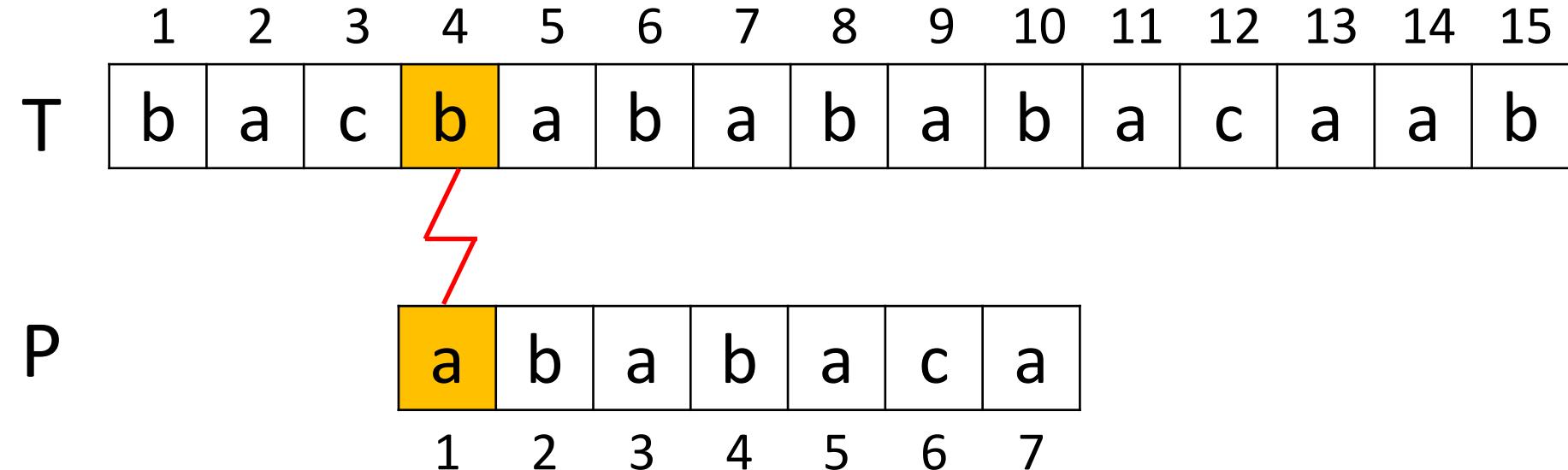
i=4

q=0

P[1] does not match with T[4].

q is increased to 1.

P will be shifted one position to the right.



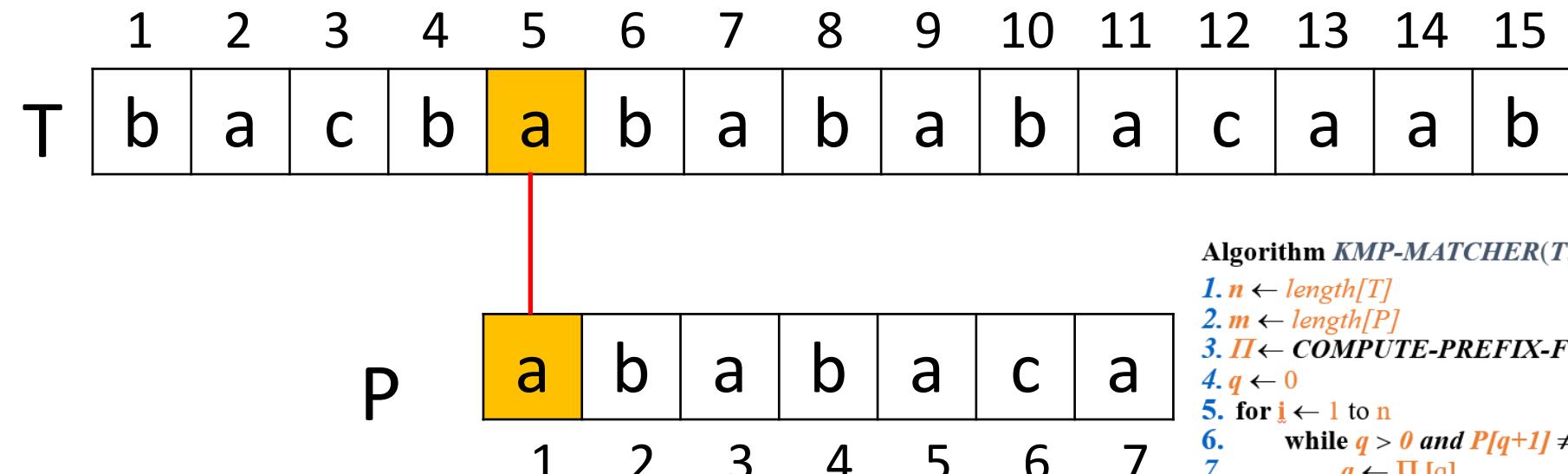
Step 5:

i=5

q=0

P[1] match with T[5].

q is increased to 1.



Algorithm KMP-MATCHER(T, P)

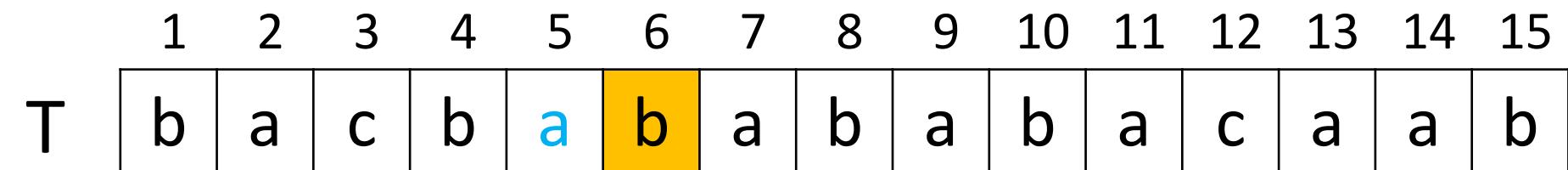
```

1.  $n \leftarrow \text{length}[T]$ 
2.  $m \leftarrow \text{length}[P]$ 
3.  $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4.  $q \leftarrow 0$ 
5. for  $i \leftarrow 1$  to  $n$ 
6.   while  $q > 0$  and  $P[q+1] \neq T[i]$ 
7.      $q \leftarrow \Pi[q]$ 
8.   if  $P[q+1] = T[i]$ 
9.      $q \leftarrow q + 1$ 
10.  if  $q = m$ 
11.    Print "Pattern occurs with shift"  $i - m$ 
12.     $q \leftarrow \Pi[q]$ 

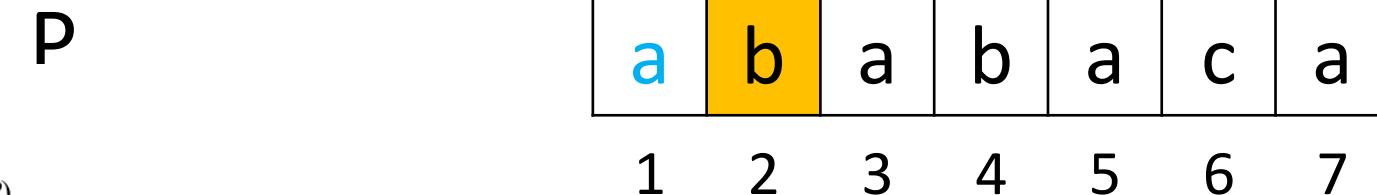
```

Step 6:

i=6
q=1

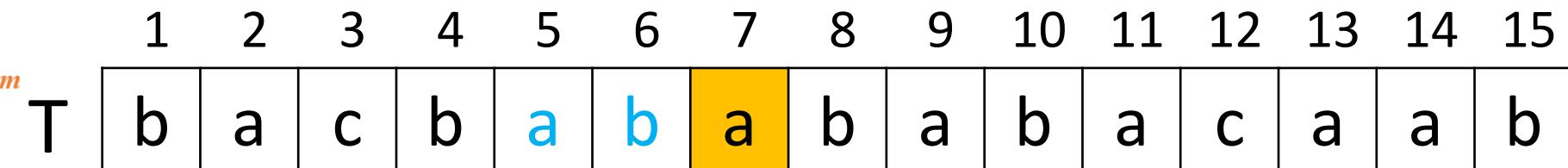


P[2] match with T[6].
q is increased to 2.



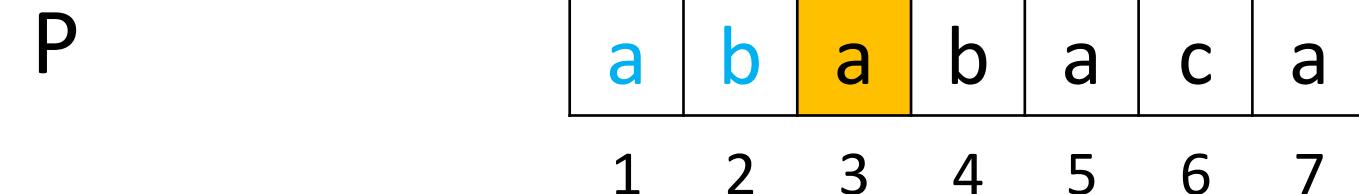
Algorithm KMP-MATCHER(T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$
5. **for** $i \leftarrow 1$ to n
6. **while** $q > 0$ and $P[q+1] \neq T[i]$
7. $q \leftarrow \Pi[q]$
8. **if** $P[q+1] = T[i]$
9. $q \leftarrow q + 1$
10. **if** $q = m$
11. Print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$



Step 7:

i=7
q=2



P[3] match with T[7].
q is increased to 3.

Step 8:

i=8

q=3

P[4] match with T[8].

q is increased to 4.

Algorithm **KMP-MATCHER(T, P)**

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$
5. for $i \leftarrow 1$ to n
6. while $q > 0$ and $P[q+1] \neq T[i]$
7. $q \leftarrow \Pi[q]$
8. if $P[q+1] = T[i]$
9. $q \leftarrow q + 1$
10. if $q = m$
11. Print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P	1	2	3	4	5	6	7
	a	b	a	b	a	c	a

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Step 9:

i=9

q=4

P[5] match with T[9].

q is increased to 5.

P	1	2	3	4	5	6	7
	a	b	a	b	a	c	a

Step 10:

i=10

q=5

P[6] does not match with T[10].

$$q = \Pi[q] = \Pi[5] = 3$$

Algorithm KMP-MATCHER(T, P)

```

1.  $n \leftarrow \text{length}[T]$ 
2.  $m \leftarrow \text{length}[P]$ 
3.  $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4.  $q \leftarrow 0$ 
5. for  $i \leftarrow 1$  to  $n$ 
6.   while  $q > 0$  and  $P[q+1] \neq T[i]$ 
7.      $q \leftarrow \Pi[q]$ 
8.   if  $P[q+1] = T[i]$ 
9.      $q \leftarrow q + 1$ 
10.  if  $q = m$ 
11.    Print "Pattern occurs with shift"  $i - m$ 
12.     $q \leftarrow \Pi[q]$ 

```

T

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P

a	b	a	b	a	c	a
1	2	3	4	5	6	7

T

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P

a	b	a	b	a	c	a
1	2	3	4	5	6	7

Π

0	0	1	2	3	0	1
1	2	3	4	5	6	7

Now P[4] match with T[10].
q is increased to 4.

Step 11:

i=11
q=4

P[5] match with T[11].
q is increased to 5.

Algorithm KMP-MATCHER(T, P)

```

1.  $n \leftarrow \text{length}[T]$ 
2.  $m \leftarrow \text{length}[P]$ 
3.  $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4.  $q \leftarrow 0$ 
5. for  $i \leftarrow 1$  to  $n$ 
6.   while  $q > 0$  and  $P[q+1] \neq T[i]$ 
7.      $q \leftarrow \Pi[q]$ 
8.   if  $P[q+1] = T[i]$ 
9.      $q \leftarrow q + 1$ 
10.  if  $q = m$ 
11.    Print "Pattern occurs with shift"  $i - m$ 
12.     $q \leftarrow \Pi[q]$ 

```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a

Step 12:

i=12
q=5

P[6] match with T[12].
q is increased to 6.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a

Step 13:

i=13

q=6

P[7] match with T[13].

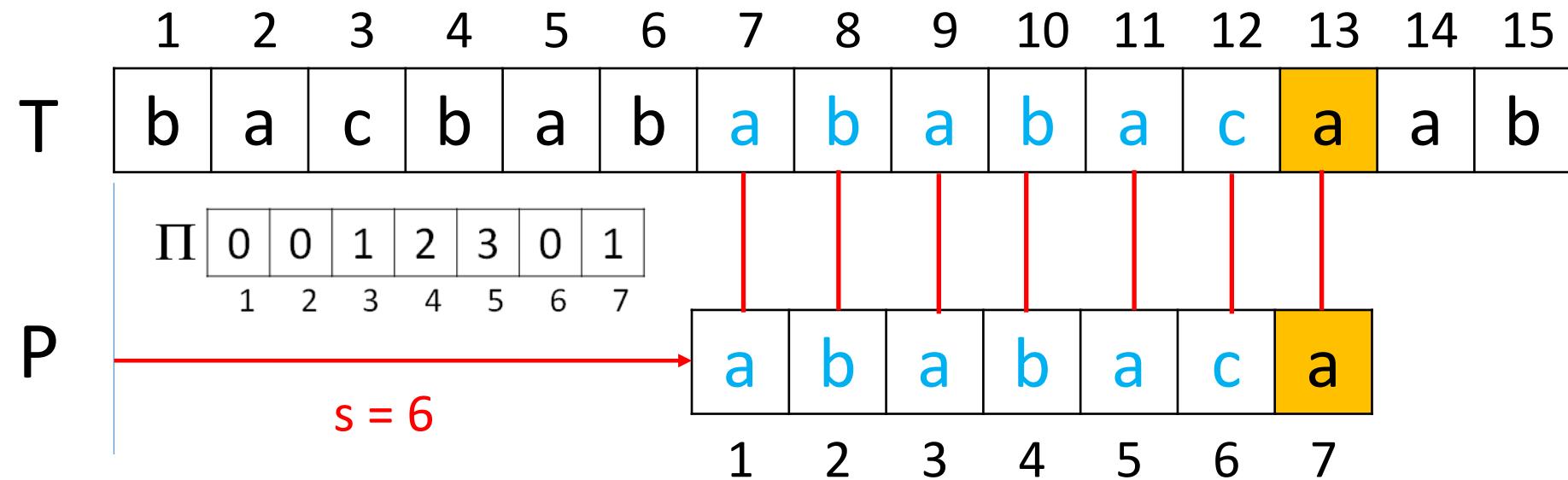
q is increased to 7.

Now q is same as m.

Prints Pattern occurs P

at shift i - m = 13 - 7

= 6. q = $\Pi[q] = \Pi[7] = 1$



Running Time

KMP-MATCHER(T, P)

- In the similar way we can find the running time is $O(n)$.
- Hence the total running time is $O(m+n) = O(n)$ as $m = O(n)$

Advantages and Disadvantages

- Advantages:
 - The running time of the KMP algorithm is optimal ($O(m + n)$), which is very fast.
 - The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.
- Disadvantages:
 - Doesn't work so well as the size of the alphabets increases. By which more chances of mismatch occurs.

Longest Common Subsequence

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: $X = \{A B C B D A B\}$,

$Y = \{B D C A B A\}$

Longest Common Subsequence:

$X = A \textcolor{green}{B} \textcolor{green}{C} \textcolor{green}{B} D \textcolor{green}{A} B$

$Y = \textcolor{green}{B} D \textcolor{green}{C} A \textcolor{green}{B} \textcolor{green}{A}$

Subsequences

Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$,

Another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is called a **subsequence** of X if there exist indices $i_1 < i_2 < \dots < i_k$ such that $x_{i_j} = z_j$

Example:

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$Z = \langle B, C, D, B \rangle$$

is a subsequence of X with corresponding indices
 $\langle 2, 3, 5, 7 \rangle$

Common Subsequences

Suppose that X and Y are two sequences.

We say that Z is a **common subsequence** of X and Y if and only if

- Z is a subsequence of X
- Z is a subsequence of Y

Ex. $X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$\langle B, C, A \rangle$ is a **common subsequence** of X and Y.

The Longest Common Subsequence Problem

Given two sequences:

$X = \langle x_1, x_2, \dots, x_m \rangle$ and

$Y = \langle y_1, y_2, \dots, y_n \rangle$, need to find a **common subsequence** of X and Y that is of **maximal length**.

Naïve Solution

- Let X be a sequence of length m and Y a sequence of length n .
- Check for every subsequence of X whether it is a subsequence of Y , and return the longest common subsequence found.
- There are 2^m subsequences of X .
- Testing a sequences whether or not it is a subsequence of Y takes $O(n)$ time.
- Thus, the **naïve algorithm** would take $O(n2^m)$ time.

Dynamic Programming

Let us try to develop a **dynamic programming** solution to the LCS problem.

Prefix

- Let $X = \langle x_1, x_2, \dots, x_m \rangle$ be a sequence.
- We denote by X_i the sequence $X_i = \langle x_1, x_2, \dots, x_i \rangle$ and call it the i^{th} prefix of X .

Ex. $X = \langle A, B, C, B, D, A, B \rangle$

$$X_4 = \langle A, B, C, B \rangle$$

X_0 is the empty sequence

How to solve LCS problem

- Characterising a LCS
- A recursive solution
- Computing the length of LCS
- Constructing an LCS

Optimal substructure

Theorem:

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$.
Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and
 Z_{k-1} is an LCS of X_m and Y_n
2. If $x_m \neq y_n$, then $z_k \neq x_m$
 $\Rightarrow Z$ is an LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$
 $\Rightarrow Z$ is an LCS of X and Y_{n-1}

Recursive solution

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i , Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

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

LCS recursive solution

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

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i, 0] = 0$

LCS recursive solution

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

- When we calculate $c[i, j]$, we consider two cases:
- **First case:** $x_i = y_j$: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

LCS recursive solution

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

- **Second case:** $x_i \neq y_j$
- As symbols don't match, our solution is not improved
 - 1) Finding an LCS of X_{i-1} and Y and
 - 2) Finding an LCS of X and Y_{j-1}

Then find the maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$

LCS Length Algorithm

LCS-Length(X, Y)

LCS Example

We'll see how LCS algorithm works on the following example:

- $X = ABCB$
- $Y = BDCAB$

What is the Longest Common Subsequence of X and Y?

$$\text{LCS}(X, Y) = BCB$$

$X = A \ B \ C \ B$

$Y = \ B \ D \ C \ A \ B$

LCS Example (0)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi						
1	A						
2	B						
3	C						
4	B						

$X = ABCB; m = |X| = 4$

$Y = BDCAB; n = |Y| = 5$

Allocate array $c[5,4]$

LCS Example (1)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

for $i = 1$ to m

$$c[i,0] = 0$$

for $j = 1$ to n

$$c[0,j] = 0$$

LCS Example (2)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	D	C	A	B		
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (3)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	
2	B	0				
3	C	0				
4	B	0				

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (4)

ABC
BDCA
B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (5)

ABC_B

BDCA_B

i	j	0	1	2	3	4	5
	Y _j	B	D	C	A	B	
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	→ 1
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (7)

ABC
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (8)

ABC
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (9)

ABC
BD CAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	2
3	C	0	1			
4	B	0				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABC
BD CAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABC
BD**C**A**B**

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (12)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (13)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
Xi	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
4	B	0	1			

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (15)

ABCB
BDCA_B

j	0	1	2	3	4	5
i	Y _j	B	D	C	A	B
X _i	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
4	B	1	1	2	2	3

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array $c[m,n]$
- So what is the running time?

$O(m*n)$

since each $c[i,j]$ is calculated in constant time, and there are $m*n$ elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

How to find actual LCS - continued

- Remember that

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

- So we can start from $c[m,n]$ and go backwards
- Look first to see if 2nd case above was true
- If not, then $c[i,j] = c[i-1, j-1]+1$, so remember $x[i]$ (because $x[i]$ is a part of LCS)
- When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

Algorithm to find actual LCS

- Here's a recursive algorithm to do this:

```
print-LCS_print(b, X, i, j)
    if i=0 or j =0
        then return
    if b[i,j] = "↖"
        then print-LCS_print(b, X, i-1, j-1)
            print xi
    else if b[i,j] = "↑"
        then print-LCS_print(b, X, i-1, j)
    else print-LCS_print(b, X, i, j-1)
```

Finding LCS

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

The diagram illustrates the construction of an LCS matrix. The matrix is filled with values representing the length of the longest common subsequence found so far. Arrows point from the bottom-right corner up and left, indicating the path of common characters between the two sequences being compared.

Finding LCS (2)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1 ← 1	1	1	1	2
3	C	0	1	1	2 ← 2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B

(this string turned out to be a palindrome)₃₈

Finding LCS (2)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1 ← 1	1	1	1	2
3	C	0	1	1	2 ← 2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B

(this string turned out to be a palindrome)₃₉

DYNAMIC PROGRAMMING

Divide-and-Conquer Vs Dynamic Programming

Divide-and-Conquer

- Refers to writing code
- Divides the problem into independent subproblems
- Does more work
- Repeatedly solving common subproblems
- Top-down approach

Dynamic Programming

- Refers to the tabular method
- Divides the problem into interdependent subproblems
- Does less work
- Solves each subproblems just once and saves its answer in a table
- Bottom-up approach

Dynamic Programming

Four-step method:

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Compute an optimal solution from computed information.

Matrix-chain Multiplication

- We have a chain (sequence) A_1, A_2, \dots, A_n of n matrices to be multiplied
 - That is, we want to compute the product $A_1 A_2 \dots A_n$
- **Fully parenthesized** – A product of matrices is fully parenthesized if it is either a **single matrix** or a product of **two parenthesized matrices** surrounded by parenthesis.
- There are many possible ways (**parenthesizations**) to compute the product

Matrix-chain Multiplication

...contd

- **Ex.** Consider the chain A_1, A_2, A_3, A_4 of 4 matrices
 - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:
 1. $(A_1(A_2(A_3A_4)))$
 2. $(A_1((A_2A_3)A_4))$
 3. $((A_1A_2)(A_3A_4))$
 4. $((A_1(A_2A_3))A_4)$
 5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication

...contd

- To compute the number of scalar multiplications necessary, we must know:
 - Algorithm to multiply two matrices
 - Matrix dimensions
- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)

Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

MATRIX-MULTIPLY($A_{p \times q}$, $B_{q \times r}$)

1. **for** $i \leftarrow 1$ **to** p
2. **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4. **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute
CNumber of scalar multiplications = pqr

Matrix-chain Multiplication

...contd

- Example: Consider three matrices $A_{10 \times 100}$, $B_{100 \times 5}$, and $C_{5 \times 50}$
 - There are 2 ways to parenthesize
 - $((AB)C) = D_{10 \times 5} \cdot C_{5 \times 50}$
 - $AB \Rightarrow 10 \cdot 100 \cdot 5 = 5,000$ scalar multiplications
 - $DC \Rightarrow 10 \cdot 5 \cdot 50 = 2,500$ scalar multiplications
 - $(A(BC)) = A_{10 \times 100} \cdot E_{100 \times 50}$
 - $BC \Rightarrow 100 \cdot 5 \cdot 50 = 25,000$ scalar multiplications
 - $AE \Rightarrow 10 \cdot 100 \cdot 50 = 50,000$ scalar multiplications
- Total: 7,500
- Total: 75,000

Matrix-chain Multiplication

...contd

- Matrix-chain multiplication problem
 - Given a chain A_1, A_2, \dots, A_n of n matrices, where for $i=1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
 - Parenthesize the product $A_1 A_2 \dots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Dynamic Programming Approach

- The structure of an optimal solution
 - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \dots A_j$
 - An optimal parenthesization of the product $A_1 A_2 \dots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
 - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

...contd

- **Key observation:** parenthesizations of the subchains $A_1A_2\dots A_k$ and $A_{k+1}A_{k+2}\dots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2\dots A_n$ is optimal (why?)
- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach

...contd

- Recursive definition of the value of an optimal solution
 - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
 - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
 - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach

...contd

- $A_{i..j} = (A_i A_{i+1} \dots A_k) \cdot (A_{k+1} A_{k+2} \dots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$
- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
for $i \leq k < j$
- $m[i, i] = 0$ for $i=1,2,\dots,n$

Dynamic Programming Approach

...contd

- But... optimal parenthesization occurs at one value of k among all possible $i \leq k < j$
- Check all these and select the best one

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

Dynamic Programming Approach

...contd

- To keep track of how to construct an optimal solution, we use a table s
- $s[i, j] = \text{value of } k \text{ at which } A_i A_{i+1} \dots A_j \text{ is split for optimal parenthesization}$
- Algorithm: next slide
 - First computes costs for chains of length $l=1$
 - Then for chains of length $l=2, 3, \dots$ and so on
 - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0\dots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

MATRIX-CHAIN-ORDER(p)

1. $n \leftarrow \text{length}[p] - 1$
2. **for** $i \leftarrow 1$ **to** n
3. $m[i, i] \leftarrow 0$
4. **for** $l \leftarrow 2$ **to** n
5. **for** $i \leftarrow 1$ **to** $n-l+1$
6. $j \leftarrow i+l-1$
7. $m[i, j] \leftarrow \infty$
8. **for** $k \leftarrow i$ **to** $j-1$
9. $q \leftarrow m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]$
10. **if** $q < m[i, j]$
11. $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. **return** m and s

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Approach

- Basically, we're checking different places to “split” our matrices by checking different values of k and seeing if they improve our current minimum value.

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table m and the split table s
- The optimal solution can be constructed from the split table s
 - Each entry $s[i, j] = k$ shows where to split the product $A_i A_{i+1} \dots A_j$ for the minimum cost

Example

- Show how to multiply this matrix chain optimally



Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Example

When $I=2$: Compute $m[1,2], m[2,3], m[3,4], m[4,5], m[5,6]$

where, $m[1,2] = 15750,$

$m[2,3] = 2650,$

$m[3,4] = 750,$

$m[4,5] = 1000,$

$m[5,6] = 5000$

Example

When $l=3$: Compute $m[1,3], m[2,4], m[3,5], m[4,6]$

$$m[1,3] = \min_{1 \leq k < 3} \{m[1,k] + m[k+1,3] + p_0 p_k p_3\}$$

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 = 7875, \\ m[1,2] + m[3,3] + p_0 p_2 p_3 = 18000 \end{cases} = 7875$$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 = 6000, \\ m[2,3] + m[4,4] + p_1 p_3 p_4 = 4375 \end{cases} = 4375$$

$$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + p_2 p_3 p_5 = 2500, \\ m[3,4] + m[5,5] + p_2 p_4 p_5 = 3750 \end{cases} = 2500$$

$$m[4,6] = \min \begin{cases} m[4,4] + m[5,6] + p_3 p_4 p_6 = 6250, \\ m[4,5] + m[6,6] + p_3 p_5 p_6 = 3500 \end{cases} = 3500$$

When $l=4$: Compute $m[1,4], m[2,5], m[3,6]$

$$m[1,4] = \min_{1 \leq k < 4} \{m[1,k] + m[k+1,4] + p_0 p_k p_4\}$$

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 = 0 + 4375 + 30 \cdot 35 \cdot 10 & = 14875, \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 15750 + 750 + 30 \cdot 15 \cdot 10 = 21000, \\ m[1,3] + m[4,4] + p_0 p_3 p_4 = 7875 + 0 + 30 \cdot 5 \cdot 10 & = 9375 \end{cases}$$
$$= 9375$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125.$$

When l=5: Compute $m[1,5], m[2,6]$

$$m[1,5] = \min_{1 \leq k < 5} \{m[1,k] + m[k+1,5] + p_0 p_k p_5$$

$$\begin{aligned} m[1,5] &= \\ \min \left\{ \begin{array}{l} m[1,1] + m[2,5] + p_0 p_1 p_5 = 0 + 7125 + 30 \cdot 35 \cdot 20 = 28125 \\ m[1,2] + m[3,5] + p_0 p_2 p_5 = 15750 + 2500 + 30 \cdot 15 \cdot 20 = 17250 \\ m[1,3] + m[4,5] + p_0 p_3 p_5 = 7875 + 1000 + 30 \cdot 5 \cdot 20 = 11875 \\ m[1,4] + m[5,5] + p_0 p_4 p_5 = 9375 + 0 + 30 \cdot 10 \cdot 20 = 15375 \end{array} \right. \\ &= 11875 \end{aligned}$$

When I=6: Compute $m[1,6]$

$$m[1,6] = \min_{1 \leq k < 6} \{m[1,k] + m[k+1,6] + p_0 p_k p_6$$

$$m[1,6] = \min \begin{cases} m[1,1] + m[2,6] + p_0 p_1 p_6 = 0 + 7125 + 30 \cdot 35 \cdot 25 = 33375 \\ m[1,2] + m[3,6] + p_0 p_2 p_6 = 15750 + 5375 + 30 \cdot 15 \cdot 25 = 32375 \\ m[1,3] + m[4,6] + p_0 p_3 p_6 = 7875 + 3500 + 30 \cdot 5 \cdot 25 = 15125 \\ m[1,4] + m[5,6] + p_0 p_4 p_6 = 9375 + 5000 + 30 \cdot 10 \cdot 25 = 21875 \\ m[1,5] + m[6,6] + p_0 p_5 p_6 = 11875 + 0 + 30 \cdot 20 \cdot 25 = 26875 \end{cases}$$
$$= 15125$$

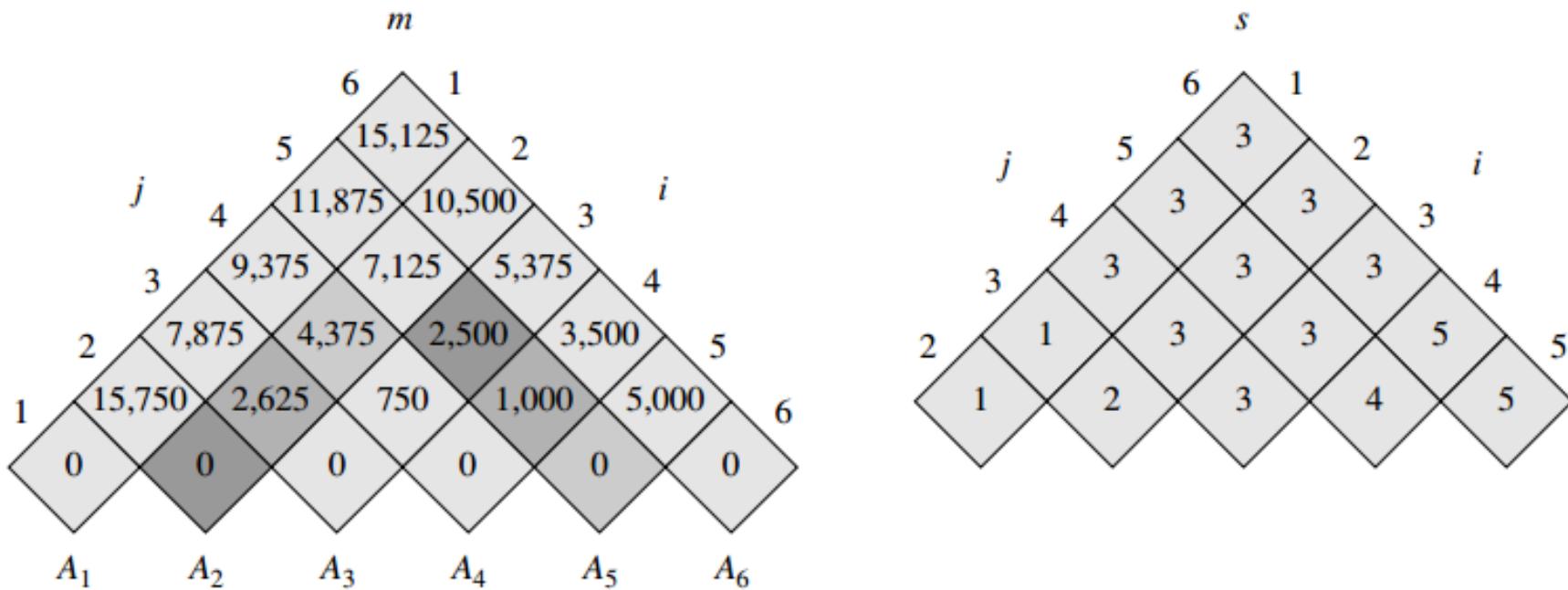


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Example

- Solution on the board
 - Minimum cost 15,125
 - Optimal parenthesization
 $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Constructing Optimal Solution

- The initial call Print-optimal-paren($S, 1, n$) print an optimal parenthesization of $\langle A_1, A_2, \dots, A_n \rangle$

Print-optimal-paren(S, i, j)

1. if $i=j$
2. then print “A” i
3. else print “(”
4. Print-optimal-paren($S, i, S[i, j]$)
5. Print-optimal-paren($S, S[i, j]+1, j$)
6. Print “) ”

Constructing Optimal Solution

The initial call Print-optimal-paren(S,1,n)

(

Print-optimal-paren(S,i,3)

Print-optimal-paren(S,4,6)

)

Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$

Merge Sort

A sorting algorithm based on divide and conquer.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p..r]$. Initially, $p=1$, and $r=n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$:

Divide: by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where boundary q is the halfway point of $A[p..r]$.

Conquer: by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

Combine: by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To achieve this step, we define P the procedure $\text{merge}(A, p, q, r)$.

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.

Merge-sort(A, p, r)

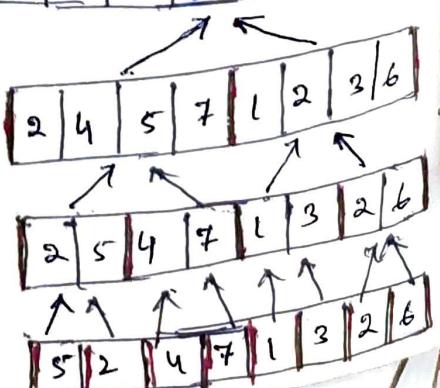
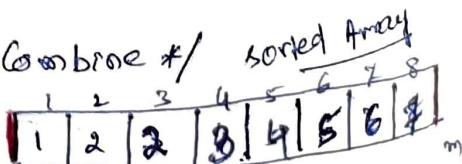
```

if  $p < r$            /* check for base case */
  then  $q \leftarrow \lfloor (p+r)/2 \rfloor$       /* divide */
        merge-sort( $A, p, q$ )    /* conquer */
        merge-sort( $A, q+1, r$ )  /*    */
        merge( $A, p, q, r$ )     /* combine */

```

Initial call: $\text{merge-sort}(A, 1, n)$

Ex = Bottom-up view for $n=8$.



Initial array

~~merging~~

internal merging in the merge merge procedure.

Input: Needs A and B as two sorted arrays.

* P < q

- * Subarray A[P:i] is sorted and returning array is sorted. By this continuing on right, smaller subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in A[P..r].

We implement it so that it takes $O(n)$ time, where $n = r - p + 1$ = the number of elements being merged.

merge(A, P, Q, R)

1. $m_1 \leftarrow Q - P + 1$
2. $m_2 \leftarrow R - Q$
3. Create arrays $L[1..m_1+1]$ and $R[1..m_2+1]$
4. for $i \leftarrow 1$ to m_1
 - do $L[i] \leftarrow A[P+i-1]$
5. for $j \leftarrow 1$ to m_2
 - do $R[j] \leftarrow A[Q+j-1]$
6. $L[m_1+1] \leftarrow \infty$
7. $R[m_2+1] \leftarrow \infty$
8. $i \leftarrow 1$
9. $j \leftarrow 1$
10. for $k \leftarrow P$ to R
 - do if $L[i] \leq R[j]$
 - then $A[k] \leftarrow L[i]$
 - else $A[k] \leftarrow R[j]$
 - 11. $i \leftarrow i + 1$
 - 12. $j \leftarrow j + 1$
13. $i \leftarrow i + 1$
14. $j \leftarrow j + 1$

18-ex.

A

8	9	10	11	12	13	14	15	16	17	-
... 2 4 5 7 1 2 3 6 - - -										

$k=1$

L	[2 4 5 7 6]
$i=1$	

R	[1 2 3 6 ∞]
$j=1$	

- (a)
- Line-1 Compute the length of subarray $A[p..q] = n_1$ — $O(1)$
- 2 - Compute " " $A[q+1..r] = n_2$ — $O(1)$
- 3 - Create 2 new arrays L and R of lengths n_1+1 and n_2+1 respectively. } $O(p)$
- 4-5 Copies the subarray $A[p..q]$ into $L[1..n_1]$ } $T(p) = O(n_1)$
- 6-7 Copies the subarray $A[q+1..r]$ into $R[1..n_2]$ } $= O(q-p+1+n_2)$
 $= O(r-p+1) = O(1)$
- 8-9 Put the sentinel (a large number) at the end of L and R, which will be used to check whether the array L and R ends. } Here $n = r-p+1$
- 10-11 Initialize the i and j to ∞ .
- 12-17 At the start of each iteration of the for loop of lines 12-17, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. And during the iterations $L[i]$ and $R[j]$, among them will be copied to $A[i]$. } $O(1)$

a	1	4	5	7	1	2	3	6	-
R									

L	[2 4 5 7 ∞]
i	

(b)

Total = $\Theta(n)$ time

-	1	2	2	1	1	1	1	-
R								

R	-	1	2	1	1	1	1	-
K								

L	[2 4 5 7 ∞]
i	

R	[1 2 3 6 ∞]
j	

(c)

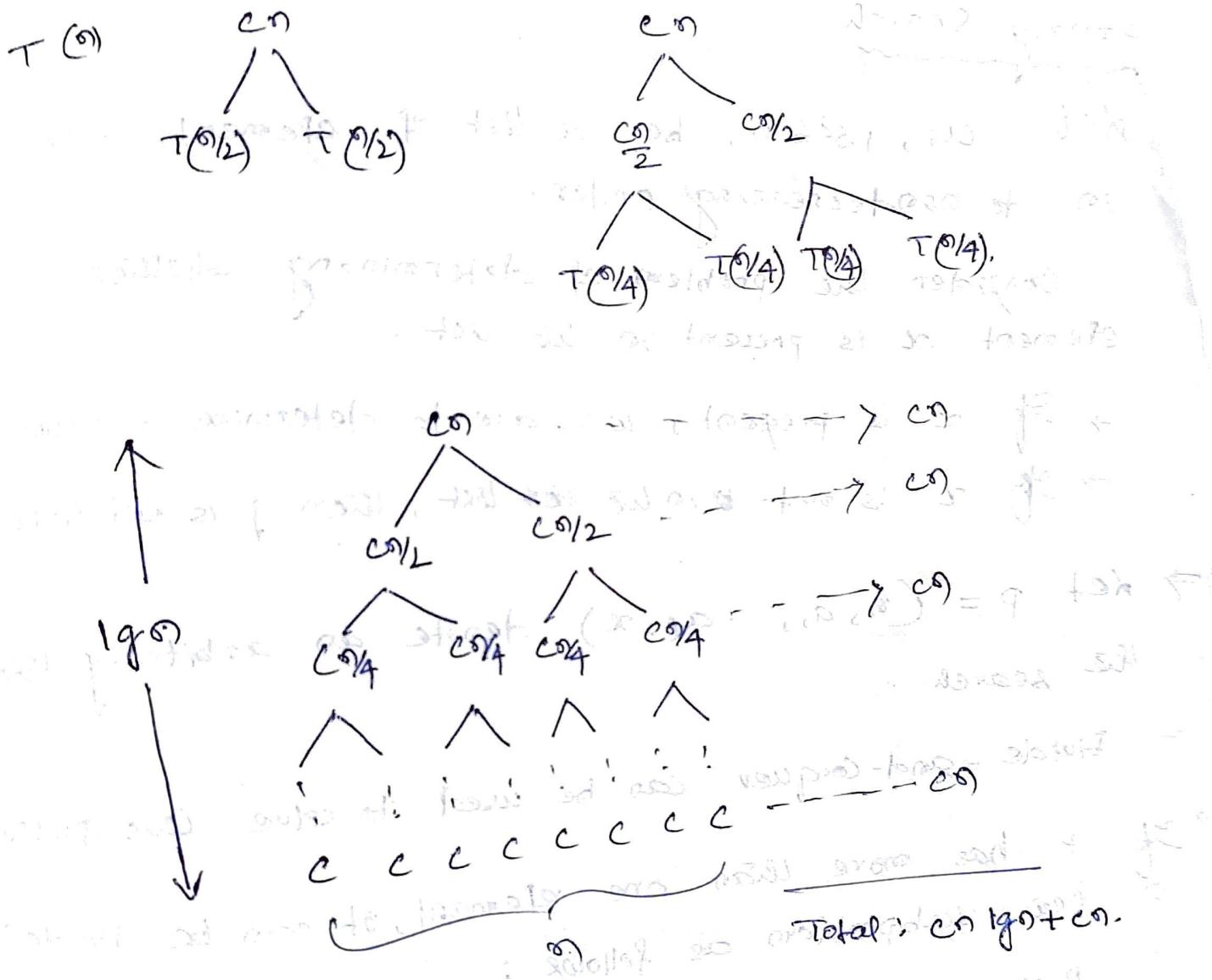
L	[2 4 5 7 ∞]
i	

R	[1 2 3 6 ∞]
j	

3	2	6

(5)

Silicon



Analyzing merge sort

Divide: Just Compute $\lg n$ as the average of $\lg 2 = \Theta(1)$

Conquer: Recursively solve 2 subproblems, each of size $n/2 = 2T(n/2) = \Theta(n)$

Combine: MERGE on an n -element subarray takes $\Theta(n)$

So, $T(n) = \begin{cases} \Theta(1), & \text{if } n=1, \\ 2T(n/2) + \Theta(n), & \text{if } n>1 \end{cases}$

Using $T(n) = \Theta(n \lg n)$

STRING MATCHING

Ajit Kumar Behera

String Matching

- Definition of string matching
- Naive string-matching algorithm
- Rabin-Karp algorithm
- Knuth-Morris-Pratt algorithm

Introduction

- What is *string matching*?
- Finding all occurrences of a *pattern* in a given *text* (or *body of text*)

Many applications

- While using editor/word processor/browser
- Login name & password checking
- Virus detection
- Header analysis in data communications
- DNA sequence analysis

TYPES OF STRING MATCHING:-

- **String matching:**

Finding all occurrences of a pattern in a text.

- Naive (Brute force) algorithm
- Rabin-Karp algorithm
- Boyer and Moore
- Knuth-Morris and Pratt

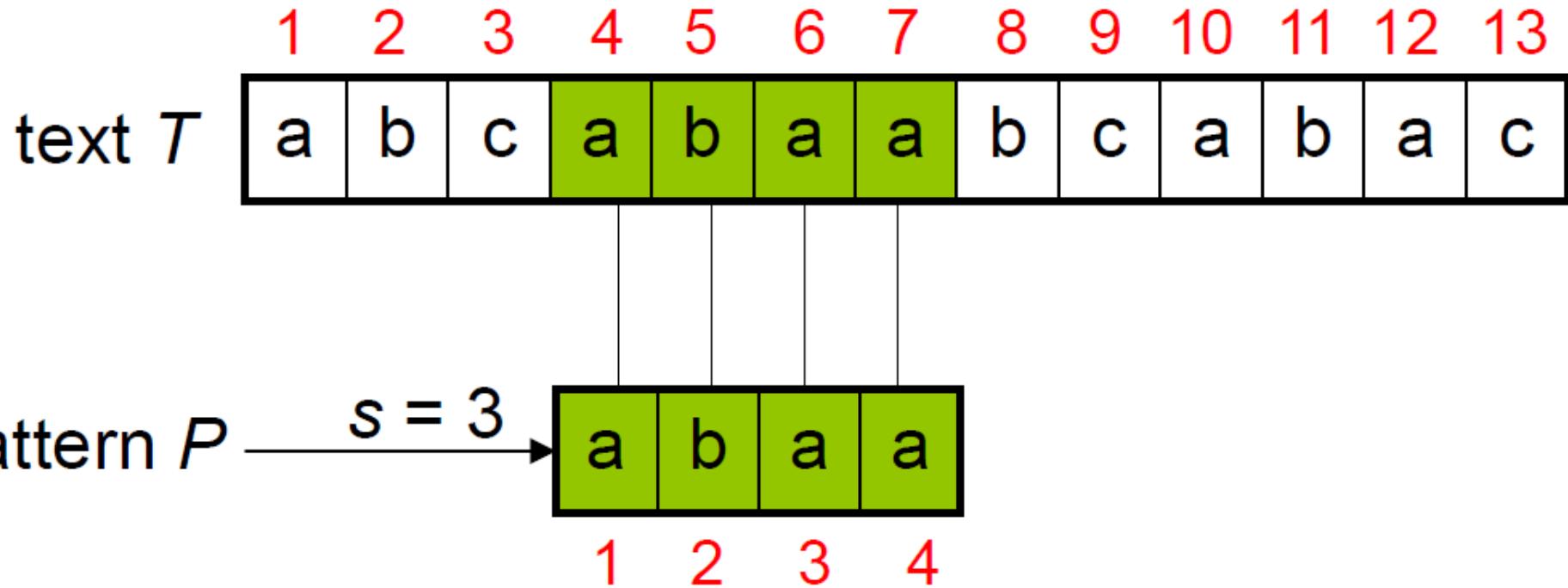
String-Matching Problem

- The *text* is in an array $T[1..n]$ of length n
- The *pattern* is in an array $P[1..m]$ of length m
- Elements of T and P are characters from a *finite alphabet* Σ
 - E.g., $\Sigma = \{0,1\}$ or
 - $\Sigma = \{a, b, \dots, z\}$
- Usually T and P are called *strings* of characters

String-Matching Problem(...contd)

- o We say that pattern P occurs with shift s in text T if:
 - a) $0 \leq s \leq n - m$ and
 - b) $T [(s+1)\dots(s+m)] = P [1..m]$
- o If P occurs with shift s in T , then s is a valid shift, otherwise s is an invalid shift
- o String-matching problem:
Finding all **valid shifts** for a given T and P

Example 1



shift $s = 3$ is a valid shift
($n=13$, $m=4$ and $0 \leq s \leq n-m$ holds)

Example 2

pattern P

1	2	3	4
a	b	a	a

text T

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	c	a	b	a	a	b	c	a	b	a	a

$$s = 3$$

a	b	a	a
---	---	---	---

$$s = 9$$

a	b	a	a
---	---	---	---

Terminology

- *Concatenation* of 2 strings x and y is xy
 - E.g., $x = \text{"Silicon"}$, $y = \text{"Institute"}$
 $\Rightarrow xy = \text{"SiliconInstitute"}$
- A string w is a *prefix* of a string x , if $x=wy$ for some string y
 - E.g., "Silicon" is a prefix of "Institute"
- A string w is a *suffix* of a string x , if $x=yw$ for some string y
 - E.g., "Institute" is a suffix of "Silicon"

Naive String-Matching Algorithm

Input: Text strings $T[1..n]$ and $P[1..m]$

Result: All valid shifts displayed

NAIVE-STRING-MATCHER (T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. **for** $s \leftarrow 0$ **to** $n-m$
4. **if** $P[1..m] = T[(s+1)...(s+m)]$
5. print “pattern occurs with shift” s

WORKING OF NAÏVE STRING MATCHING

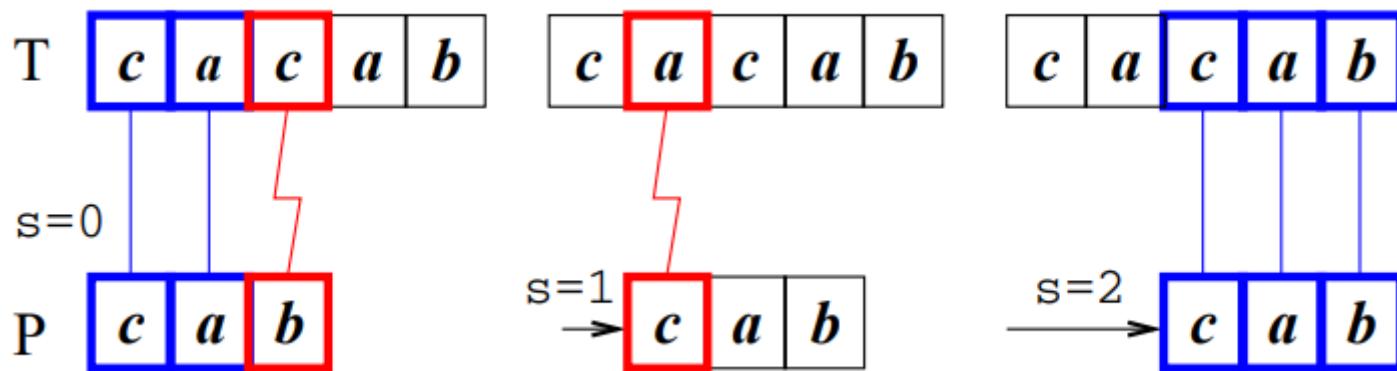
- The naive string-matching procedure can be interpreted graphically as sliding a “template” containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text.

Naive String-Matching Algorithm

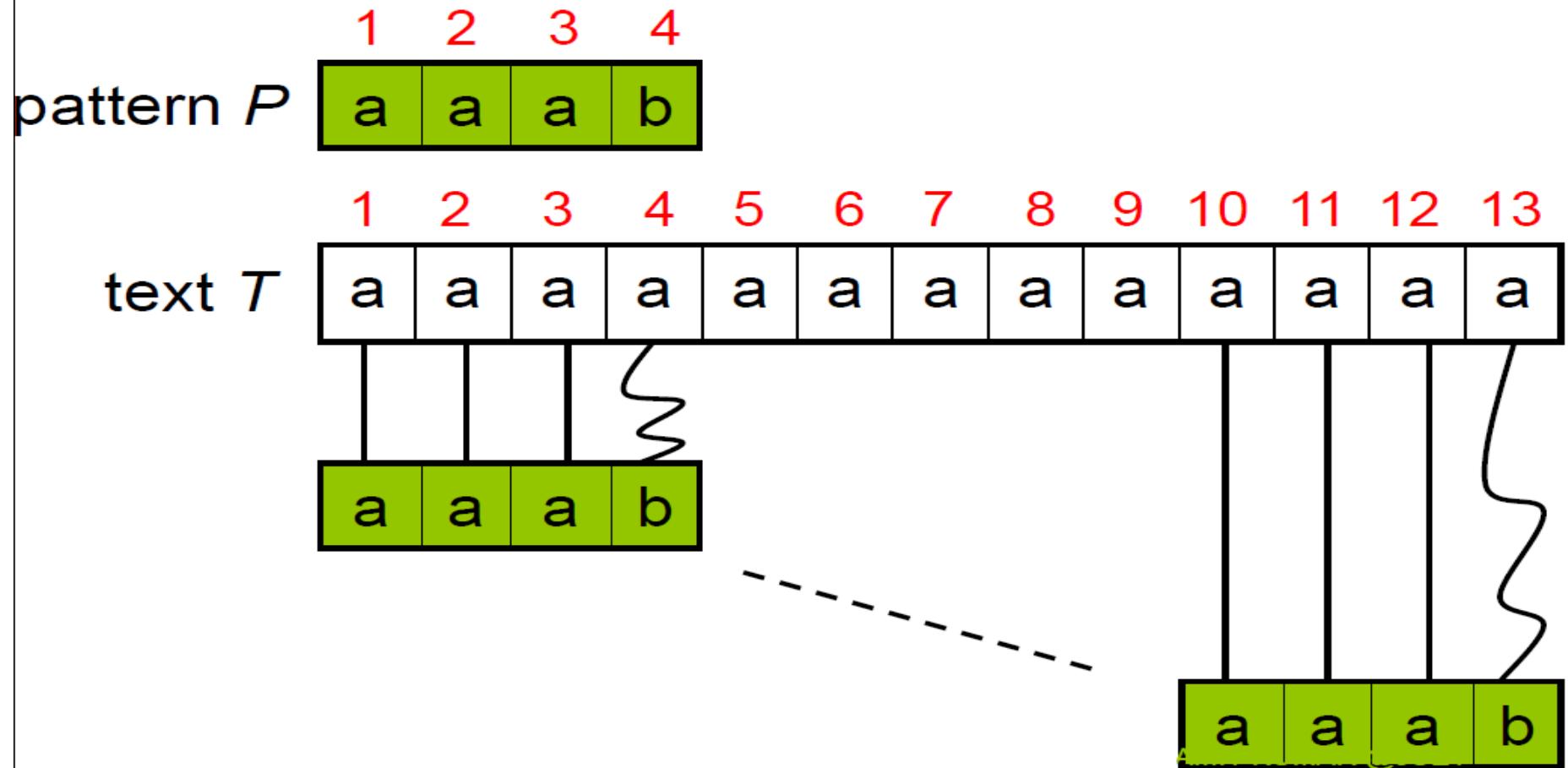
Input: Text strings $T[1..n]$ and $P[1..m]$

Result: All valid shifts displayed

Example:



Analysis: Worst-case Example



Worst-case Analysis

- There are m comparisons for each shift in the worst case
- There are $n-m+1$ shifts
- So, the worst-case running time is
$$\theta((n-m+1)m)$$
 - We have $(13-4+1) \cdot 4$ comparisons in total
 - Naive method is inefficient because information from a shift is not used again

ADVANTAGES:-

- No preprocessing phase required because the running time of NAIVE-STRING-MATCHER is equal to its matching time
- No extra space are needed.
- Also, the comparisons can be done in any order.

QUESTION???

Consider a situation where all characters of pattern are different. Can we modify [the original Naive String Matching algorithm](#) so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

ANSWER:-

In the original Naive String matching algorithm, we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1.

When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts.

Rabin-Karp String Matching Algorithm

- Rabin-Karp algorithm uses $\theta(m)$ preprocessing and its worst case running time is $\theta((n-m+1)m)$
- However its average-case running time is better

Definition of Rabin-Karp

- A string search algorithm which compares a string's **hash values**, rather than the strings themselves.
- For efficiency, the **hash value** of the **next position** in the text is easily computed from the **hash value** of the **current position**.

How Rabin-Karp works

- Let characters in both arrays T and P be digits in radix- Σ notation. ($\Sigma = (0,1,\dots,9)$)
- Let p be the value of the characters in P
- Choose a prime number q such that fits within a computer word to speed computations.
- Compute $(p \bmod q)$
 - The value of $p \bmod q$ is what we will be using to find all matches of the pattern P in T .

How Rabin-Karp works (continued)

- Compute $(T[s+1, \dots, s+m] \text{ mod } q)$ for $s = 0 \dots n-m$
- Test against P only those sequences in T having the same **(mod q)** value
- $(T[s+1, \dots, s+m] \text{ mod } q)$ can be **incrementally computed** by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Rabin-Karp string matching Algorithm

- Given a pattern $P[1 \dots m]$, we let p be its corresponding decimal value.
- Given a string $T[1 \dots n]$, let t_s be the decimal value of the length- m substring $T[s + 1 \dots s + m]$.
- $t_s = p \Leftrightarrow T[s + 1 \dots s + m] = P[1 \dots m]$
- If we compute p in time $\theta(m)$ and all t_s in $\theta((n-m+1))$, then we determine all valid shift s in $\theta(m) + \theta((n-m+1)) = \theta(n)$ by comparing p with each of t_s 's.

Rabin-Karp string matching Algorithm

- Compute p in $\theta(m)$ using **Horner's rule** as:
$$p = P[m] + 10(P[m - 1] + 10(P[m - 2] + \dots + 10(P[2] + 10P[1])))$$
- Value of t_0 is computed from $T[1 \dots n]$ in time $\theta(m)$
- To compute t_1, t_2, \dots, t_{n-m} in time $\theta(n-m)$, it is suffices to observe that t_{s+1} can be computed from t_s in constant time as:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Example

- Let $m=5$, $t_s=31415$
- Remove the high order digit $T[s+1]= 3$ and bring new low-order digit (suppose $T[s+5+1]= 2$) to obtain

$$\begin{aligned}t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\&= 14152\end{aligned}$$

A Rabin-Karp example

- Given $T = 31415926535$ and $P = 26$
- We choose $q = 11$
- $P \bmod q = 26 \bmod 11 = 4$

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$31 \bmod 11 = 9$ not equal to 4

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$14 \bmod 11 = 3$ not equal to 4

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$41 \bmod 11 = 8$ not equal to 4

Rabin-Karp example continued

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$15 \bmod 11 = 4$ equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$59 \bmod 11 = 4$ equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$92 \bmod 11 = 4$ equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$26 \bmod 11 = 4$ equal to 4 -> an exact match!!

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$65 \bmod 11 = 10$ not equal to 4

Rabin-Karp example continued

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$53 \bmod 11 = 9$ not equal to 4

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

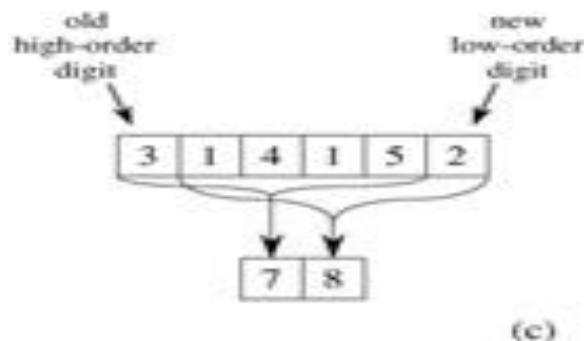
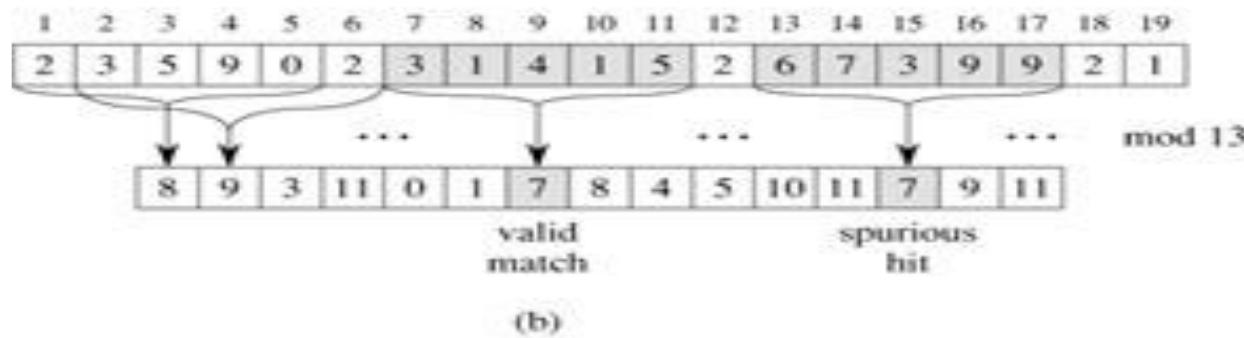
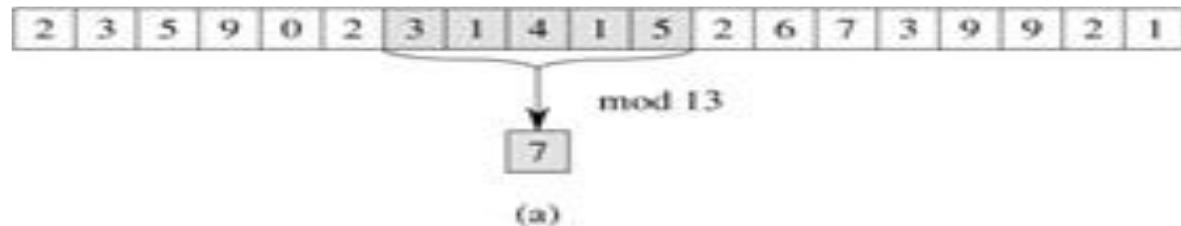
$35 \bmod 11 = 2$ not equal to 4

As we can see, when a match is found, further testing is done to insure that a match has indeed been found.

Rabin-Karp example continued

- If P contains m character, then assuming that each arithmetic operation on p takes “constant time” is undesirable.
- So, $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \text{ mod } q$,
where, $h = d^{m-1}$ and $\Sigma = (0, 1, \dots, d-1)$

Rabin-Karp string matching Algorithm



$$\begin{aligned}14152 &= (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\&= (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\&= 8 \pmod{13}\end{aligned}$$

Rabin-Karp string matching Algorithm

RABIN-KARP-MATCHER (T, P, d, q)

```
1  $n \leftarrow \text{length}[T]$ 
2  $m \leftarrow \text{length}[P]$ 
3  $h \leftarrow d^{m-1} \bmod q$ 
4  $p \leftarrow 0$ 
5  $t_0 \leftarrow 0$ 
6 for  $i \leftarrow 1$  to  $m$            ▷ Preprocessing.
7   do  $p \leftarrow (dp + P[i]) \bmod q$ 
8    $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ 
9 for  $s \leftarrow 0$  to  $n - m$       ▷ Matching.
10  do if  $p = t_s$ 
11    then if  $P[1 \dots m] = T[s + 1 \dots s + m]$ 
12      then print "Pattern occurs with shift"  $s$ 
13    if  $s < n - m$ 
14      then  $t_{s+1} \leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$ 
```

THANK YOU