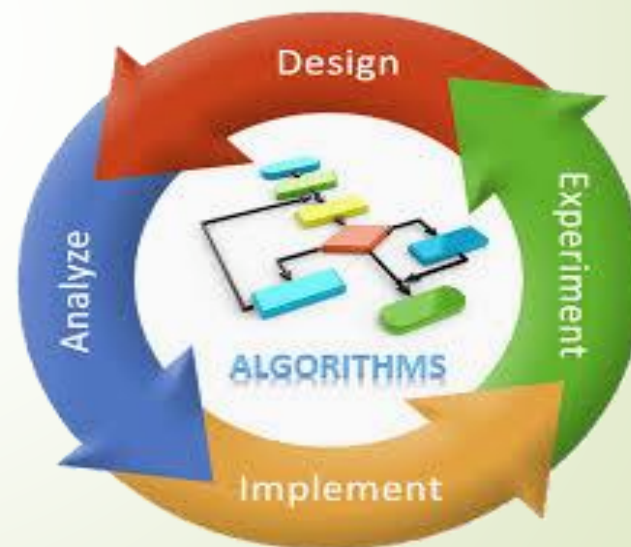




# DAA

## DESIGN AND ANALYSIS OF ALGORITHMS

**Mohd Nawazuddin**  
**Asst.Prof**



# SYLLABUS

## UNIT – I

**INTRODUCTION:** Algorithm, Performance Analysis-Space complexity, Time complexity, Asymptotic Notations- Big oh notation, Omega notation, Theta notation and Little oh notation. Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort, Strassen's matrix multiplication.

## UNIT – II

**Disjoint Sets:** Disjoint set operations, union and find algorithms  
**Backtracking:** General method, applications, n-queen's problem, sum of subsets problem, graph coloring

## UNIT - III

**Dynamic Programming:** General method, applications-Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Traveling sales person problem, Reliability design.

## UNIT – IV

**Greedy method:** General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

## UNIT - V

**Branch and Bound:** General method, applications - Travelling sales person problem, 0/1 knapsack problem - LC Branch and Bound solution, FIFO Branch and Bound solution. NP-Hard and NP-Complete problems: Basic concepts, non deterministic algorithms, NP - Hard and NP-Complete classes, Cook's theorem.

## TEXT BOOKS

1. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahni and Rajasekharan, 3<sup>rd</sup> Edition University Press.



# REFERENCES

- Design and Analysis of Algorithms, Aho, Ullman and, Pearson education.
  - Introduction to Algorithms, second edition, T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stien, PHI Pvt . Ltd./Pearson Education.
  - Algorithm Design; Foundations, Analysis and Internet Examples, M.T. Goodrich and R. Tamassia, John Wiley and sons.
-

# INTRODUCTION

- The word algorithm comes from the name of the person author -**Abu Jafar Mohammed Ibn Musa Al khowarizmi** who wrote A text book entitled-”**Algorithmi de numero indorum**” Now term” Algorithmi ”in the title of the book led to the term **Algorithm**.
- An algorithm is an effective method for finding out the solution for a given problem. It is a sequence of instruction That conveys the method to address a problem
- **Algorithm** : Step by step procedure to solve a computational problem is called Algorithm.
- or
- An Algorithm is a step-by-step plan for a computational procedure that possibly begins with an input and yields an output value in a finite number of steps in order to solve a particular problem.

# INTRODUCTION

- An algorithm is a set of steps of operations to **solve a problem** performing calculation, data processing, and automated reasoning tasks.
- An algorithm is an efficient method that can be expressed within finite amount of **Time and space**.
- The important aspects of algorithm design include creating an **efficient algorithm** to solve a problem in an efficient way using minimum time and space.
- To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient.

# PROPERTIES OF ALGORITHM

**TO EVALUATE AN ALGORITHM WE HAVE TO SATISFY THE FOLLOWING CRITERIA:**

- 1.INPUT:** The Algorithm should be given **zero** or more input.
- 2.OUTPUT:** **At least one** quantity is produced. For each input the algorithm produced value from specific task.
- 3.DEFINITENESS:** Each instruction is clear and **unambiguous**.
- 4.FINITENESS:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a **finite number** of steps.
- 5.EFFECTIVENESS:** Every instruction must **very basic** so that it can be carried out, in principle, by a person using only pencil & paper.



## ALGORITHM (CONTD...)

- A well-defined **computational procedure** that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.
- Written in a **pseudo code** which can be implemented in the language of programmer's choice.

**PSEUDO CODE:** A notation resembling a simplified programming language, used in program design.

# How To Write an Algorithm

Step-1:start

Step-2:Read a,b,c

Step-3:if a>b

    if a>c

        print a is largest

    else

        if b>c

            print b is largest

        else

            print c is largest

Step-4 : stop

Step-1: start

Step-2: Read a,b,c

Step-3:if a>b then go to step 4

        otherwise go to step 5

Step-4:if a>c then

    print a is largest otherwise

    print c is largest

Step-5: if b>c then

    print b is largest otherwise

    print c is largest

step-6: stop



# Differences

## Algorithm

1. At design phase
2. Natural language
3. Person should have Domain knowledge
4. Analyze

## Program

1. At Implementation phase
  2. written in any programming language
  3. Programmer
  4. Testing
-

# ALGORITHM SPECIFICATION

Algorithm can be described (Represent) in four ways.

## 1. Natural language like English:

When this way is chosen, care should be taken, we should ensure that each & every statement is definite.

(no ambiguity)

## 2. Graphic representation called flowchart:

This method will work well when the algorithm is small & simple.

## 3. Pseudo-code Method:

In this method, we should typically describe algorithms as program, which resembles language like Pascal & Algol (Algorithmic Language).

## 4. Programming Language:

we have to use programming language to write algorithms like C, C++, JAVA etc.

## PSEUDO-CODE CONVENTIONS

1. Comments begin with `//` and continue until the end of line.
2. Blocks are indicated with matching braces `{` and `}`.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.

```
node= record
    {
    data type 1 data 1;
    data type n data n;
    node *link;
    }
```

4. There are two Boolean values **TRUE** and **FALSE**.

Logical Operators

**AND, OR, NOT**

Relational Operators

**<, <=, >, >=, =, !=**

5. Assignment of values to variables is done using the assignment statement.

```
<Variable>:= <expression>;
```

6. Compound data types can be formed with records. Here is an example,

Node. Record

```
{  
  data type – 1  data-1;  
  .  
  .  
  .  
  data type – n  data – n;  
  node * link;  
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with  $\rightarrow$  and period.

Contd...

7. The following looping statements are employed.

For, while and repeat-until While Loop:

While < condition > do

```
{  
    <statement-1>  
    ..  
    ..  
    <statement-n>  
}
```

**For Loop:**

For variable: = value-1 to value-2 step step do

```
{  
    <statement-1>  
    .  
    .  
    .  
    <statement-n>  
}
```

## repeat-until:

```
repeat
  <statement-1>
  .
  .
  .
  <statement-n>
until<condition>
```

8. A conditional statement has the following forms.

- If <condition> then <statement>
- If <condition> then <statement-1>  
Else <statement-1>





## Case statement:

```
Case
{
    : <condition-1> : <statement-1>
    .
    .
    .
    : <condition-n> : <statement-n>
    : else : <statement-n+1>
}
```

9. Input and output are done using the instructions **read & write**. No format is used to specify the size of input or output quantities

Contd...

10. There is only one type of procedure: Algorithm, the heading takes the form,  
Algorithm Name (Parameter lists)

consider an example, the following algorithm finds & returns the maximum of n given numbers:

```
1.  algorithm Max(A,n)
2.  // A is an array of size n
3.  {
4.  Result := A[1];
5.  for i:= 2 to n do
6.  if A[i] > Result then
7.  Result :=A[i];
8.  return Result;
9.  }
```



## Issue in the study of algorithm

1. How to create an algorithm.
2. How to validate an algorithm.
3. How to analyses an algorithm
4. How to test a program.

1 .How to create an algorithm: To create an algorithm we have following design technique

- a) Divide & Conquer
- b) Greedy method
- c) Dynamic Programming
- d) Branch & Bound
- e) Backtracking

**2.How to validate an algorithm:** Once an algorithm is created it is necessary to show that it computes the correct output for all possible legal input , this process is called algorithm validation.

**3.How to analyses an algorithm:** Analysis of an algorithm or performance analysis refers to task of determining how much computing Time & storage algorithms required.

- a) Computing time-Time complexity: Frequency or Step count method
- b) Storage space- To calculate space complexity we have to use number of input used in algorithms.

**4.How to test the program:** Program is nothing but an expression for the algorithm using any programming language. To test a program we need following

- a) Debugging: It is processes of executing programs on sample data sets to determine whether faulty results occur & if so correct them.
- b) Profiling or performance measurement is the process of executing a correct program on data set and measuring the time & space it takes to compute the result.


# ANALYSIS OF ALGORITHM

## PRIORI

1. Done priori to run algorithm on a specific system
2. Hardware independent
3. Approximate analysis statistics of an hardware
4. Dependent on no of time statements are executed not do posteriori

## POSTERIORI


1. Analysis after running on a it on system.
2. Dependent on hardware
3. Actual algorithm
4. They do analysis



**Problem:** Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Pseudo Approach


1. Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
  2. FOR EACH Student PRESENT DO the following:  
Increase the **Count** by One
  3. Then Subtract **Count** from **total** and store the result in **absent**
  4. Display the number of absent students
-



**Problem:** Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Algorithmic Approach:

- 1.Count  $\leftarrow$  0, absent  $\leftarrow$  0, total  $\leftarrow$  60
  - 2.REPEAT till all students counted  
    Count  $\leftarrow$  Count + 1
  - 3.absent  $\leftarrow$  total - Count
  - 4.Print "Number absent is:" , absent
-

- 
1. To understand the basic idea of the problem.
  2. To find an approach to solve the problem.
  3. To improve the efficiency of existing techniques.
  4. To understand the basic principles of designing the algorithms. To compare the performance of the algorithm with respect to other techniques.
  6. It is the best method of description without describing the implementation detail.
  7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
  8. A good design can produce a good solution.
  9. To understand the flow of the problem.

---

## **Need of Algorithm**



# PERFORMANCE ANALYSIS

**Performance Analysis:** An algorithm is said to be **efficient and fast** if it take **less time** to execute and consumes **less memory space** at run time is called Performance Analysis.

## 1. SPACE COMPLEXITY:

The space complexity of an algorithm is the amount of **Memory Space** required by an algorithm during course of execution is called space complexity .There are three types of space

- a) Instruction space :executable program
- b) Data space: Required to store all the constant and variable data space.
- c) Environment: It is required to store environment information needed to resume the suspended space.

## 2. TIME COMPLEXITY:

The time complexity of an algorithm is the total amount of **time required** by an algorithm to complete its execution.

---



# Space complexity

Now there are two types of space complexity

a) Constant space complexity

b) Linear(variable)space complexity

---

1. Constant space complexity: A fixed amount of space for all the input values.

Example : int square(int a)

{

return a\*a;

}

Here algorithm requires fixed amount of space for all the input values.

---

2. Linear space complexity: The space needed for algorithm is based on size.

- Size of the variable 'n' = 1 word
- Array of a values = n word
- Loop variable = 1 word
- Sum variable = 1 word

Example:

```
int sum(int A[],int n)
{
    n
    int sum=0,i;    1
    for (i=0;i<n;i++) 1
    Sum=sum+A[i]; 1
    Return sum;
}
```

---

Ans :  $1+n+1+1 = n+3$  words

# 1. Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to complete.

Space needed by an algorithm is given by

$$S(P) = C(\text{fixed part}) + Sp(\text{Variable part})$$

**fixed part:** independent of instance characteristics.

eg: space for simple variables, constants etc

**Variable part:** Space for variables whose size is dependent on particular problem instance.



## Examples:

1. Algorithm sum(a,,b,c)

{

a=10;            a-1

b=20;            b-1

c=a+b;           c-1

}

$s(p) = c + sp$

$3 + 0 = 3$

$O(n) = 3$

---

## 2. algorithm sum(a,n)

```
{  
total=0;- 1  
Fori=1 to n do -1,1  
Total=total+a[i]--n  
Return total
```

---

### Algorithm-1

Algorithm abc(a,b,c)

```
{  
return a+b*c+(a+b-c)/(a+b) +4.0;  
}
```

a → 1

b → 1

c → 1

-----  
≥ 3 units  
-----

### Algorithm-2

- Algorithm sum(a,n)
- {
- s=0.0;
- for i=1 to n do
- s= s+a[i];
- return s;
- }

i,n,s → 1 unit each

a → n units

-----  
≥ n+1 units  
-----

### Algorithm-3

Algorithm RSum(a,n)

```
{  
if(n≤0) then return 0.0;  
else return Rsum(a,n-1)+a[n];  
}
```

DAA

Rsum(a,n) → 1(a[n])+1(n)+1(return)=3units

Rsum(a,n-1) → 1(a[n-1])+1(n)+1(return)

·  
·  
·

Rsum(a,n-n) → 1(a[n-n])+1(n)+1(return)

-----  
Total → ≥ 3(n+1) units



## 2. Time Complexity:


The time complexity of an algorithm is the amount of computer time it needs to run to complete.

$T(P) = \text{compile time} + \text{execution time}$

$T(P) = t_p$  (execution time)

### Step count:

- For algorithm heading  $\rightarrow 0$
- For Braces  $\rightarrow 0$
- For expressions  $\rightarrow 1$
- for any looping statements  $\rightarrow$  no. of times the loop is repeating.




1. **Constant time complexity** : If a program required **fixed** amount of time for all input values is called Constant time complexity .

Example : `int sum(int a , int b)`

```
{  
  return a+b;  
}
```





2. **Linear time complexity:** If the input values are increased then the time complexity will **changes**.

- comments = 0 step
  - Assignment statement = 1 step
  - condition statement = 1 step
  - loop condition for n times =  $n+1$  steps
  - body of the loop = n steps
-

Example : `int sum(int A[],int n)`

```
{  
    int sum=0,i;  
    for (i=0;i<n;i++)  
        sum=sum+A[i];  
    return sum;  
}
```

cost

repetition

total

1

1

1

1+1+1

1+(n+1)+n

2n+2

2

n

2n

1

1

1

4n+4

---

### Algorithm-1

```
1. Algorithm abc(a,b,c)
2. {
3. return a+b*c+(a+b-c)/(a+b) +4.0;
4. }
```

→0

→0

→1

→0

-----  
1 unit  
-----

### Algorithm-2

```
1. Algorithm sum(a,n)
2. {
3. s=0.0;
4. for i=1 to n do
5. s= s+a[i];
6. return s;
7. }
```

→0

→0

→1

→n+1

→n

→1

→0

-----  
2n+3

### Algorithm-3

```
Algorithm RSum(a,n)
{
if(n≤0) then return 0.0;
else return Rsum(a,n-1)+a[n];
}
```

DAA

$$T(n) = 2 \quad \text{if } n=0$$
$$= 2 + T(n-1) \quad \text{if } n>0$$

$$T(n) = 2 + T(n-1)$$
$$= 2 + (2 + T(n-2)) = 2*2 + T(n-2)$$
$$= 2*2 + (2 + T(n-3)) = 2*3 + T(n-3)$$
$$\vdots$$
$$\vdots$$
$$= 2*n + T(n-n) = 2n + T(0)$$
$$T(n) = 2n + 2$$

# TIME COMPLEXITY

The time  $T(p)$  taken by a program  $P$  is the sum of the compile time and the run time(execution time)

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. for i=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
<i>Total</i>			2n+3

# KINDS OF ANALYSIS

## 1. Worst-case: (usually)

- $T(n)$  = maximum time of algorithm on any input of size  $n$ .

## 2. Average-case: (sometimes)

- $T(n)$  = expected time of algorithm over all inputs of size  $n$ .
- Need assumption of statistical distribution of inputs.

## 3. Best-case:

- $T(n)$  = minimum time of algorithm on any input of size  $n$ .

## COMPLEXITY:

Complexity refers to the rate at which the storage time grows as a function of the problem size



# Analysis of an Algorithm

- The goal of analysis of an algorithm is to compare algorithm in **running time** and also **Memory management**.
- Running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

Running time of an algorithm depends on

- 1.Speed of computer
- 2.Programming language
- 3.Compiler and translator

**Examples: binary search, linear search**

---



## ASYMPTOTIC ANALYSIS:

- Expressing the complexity in term of **its relationship** to know function. This type analysis is called asymptotic analysis.
  - The main idea of Asymptotic analysis is to have a measure of **efficiency of an algorithm** , that **doesn't** depends on
    1. Machine constants.
    2. Doesn't require algorithm to be implemented.
    3. Time taken by program to be prepare.
-

# ASYMPTOTIC NOTATION

**ASYMPTOTIC NOTATION:** The mathematical way of representing the Time complexity.

The notation we use to describe the asymptotic **running time of an algorithm** are defined **in terms of functions** whose domains are the set of natural numbers.

**Definition :** It is the way to describe the behavior of functions in the limit or without bounds.

**Asymptotic growth:** The rate at which the function grows...

“growth rate” is the complexity of the function or the amount of resource it takes up to compute.



# Classification of growth

1. Growing with the **same** rate.
  2. Growing with the **slower** rate.
  3. Growing with the **faster** rate.
-

They are 3 asymptotic notations are mostly used to represent time complexity of algorithm.

1. Big oh ( $O$ ) notation
  2. Big omega ( $\Omega$ ) notation
  3. Theta ( $\Theta$ ) notation
  4. Little oh notation
  5. Little omega ( $\omega$ ) notation
-

**1. Big oh (O) notation** : Asymptotic “less than” (slower rate). This notation mainly represent upper bound of algorithm run time.

Big oh (O) notation is useful to calculate maximum amount of time of execution.

By using Big-oh notation we have to calculate worst case time complexity.

**Formula** :  $f(n) \leq c g(n)$                        $n \geq n_0, c > 0, n_0 \geq 1$

Definition: Let  $f(n), g(n)$  be two non negative (positive) function

now the  $f(n) = O(g(n))$  if there exist two positive constant  $c, n_0$  such that

$f(n) \leq c.g(n)$  for all value of  $n > 0$  &  $c > 0$

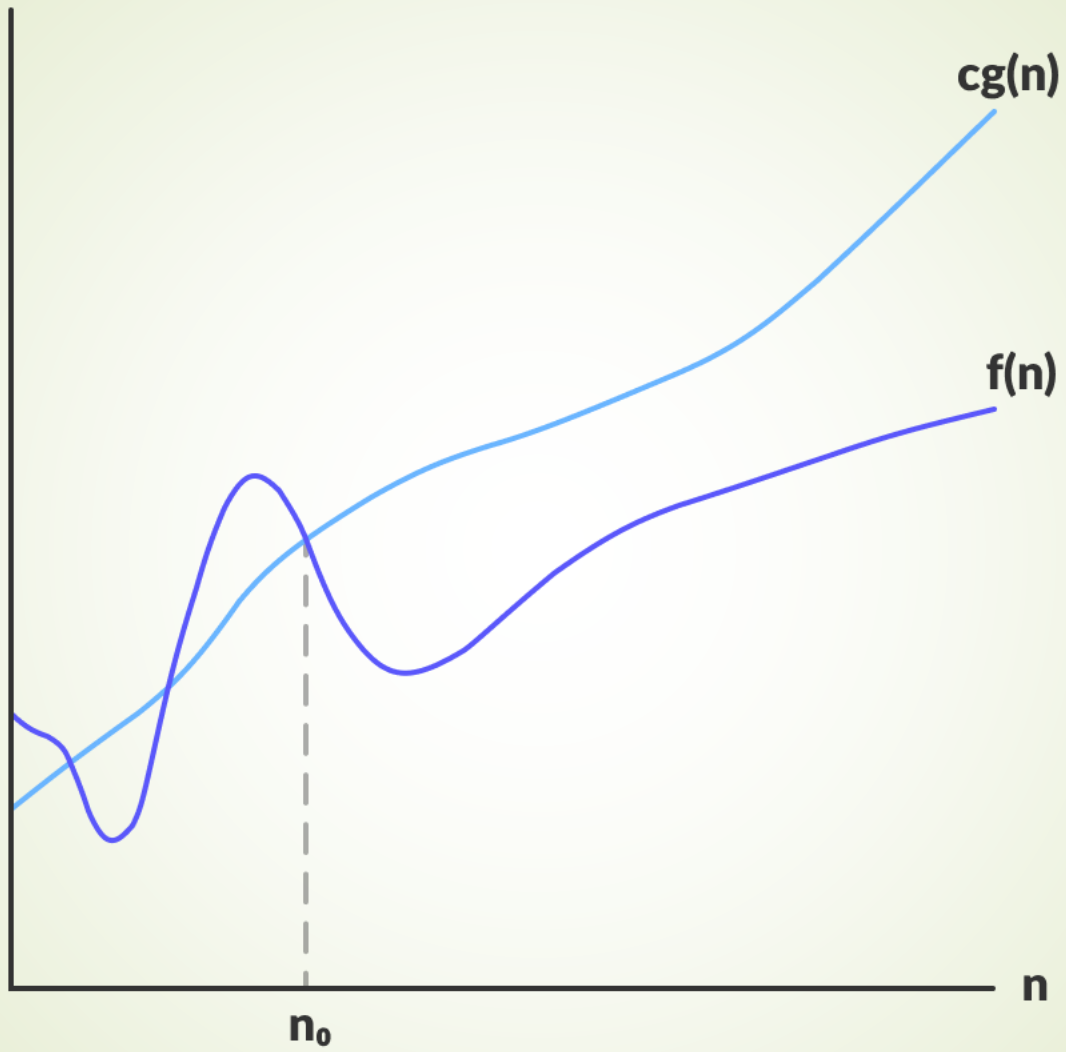
---

# 1. Big O-notation

- ❖ For a given function  $g(n)$ , we denote by  $O(g(n))$  the set of functions

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- ❖ We use O-notation to give an asymptotic upper bound of a function, to within a constant factor.
- ❖  $f(n) = O(g(n))$  means that there exists some constant  $c$  s.t.  $f(n)$  is always  $\leq cg(n)$  for large enough  $n$ .



# Examples

Example :  $f(n)=2n+3$  &  $g(n)=n$

Formula :  $f(n) \leq c \cdot g(n)$        $n \geq n_0, c > 0, n_0 \geq 1$

$f(n)=2n+3$  &  $g(n)=n$

Now  $3n+2 \leq c \cdot n$

$3n+2 \leq 4 \cdot n$

Put the value of  $n=1$

$5 \leq 4$  false

$N=2$   $8 \leq 8$  true    now  $n_0 > 2$  For all value of  $n > 2$  &  $c=4$

now  $f(n) \leq c \cdot g(n)$

$3n+2 \leq 4n$  for all value of  $n > 2$

Above condition is satisfied this notation takes maximum amount of time to execute .so that it is called worst case complexity.

---



## 2.Ω-Omega notation

**Ω-Omega notation** : Asymptotic “greater than” (faster rate).

It represent Lower bound of algorithm run time.

By using Big Omega notation we can calculate minimum amount of

time. We can say that it is best case time complexity.

**Formula** :  $f(n) \geq c g(n)$        $n \geq n_0, c > 0, n_0 \geq 1$

where c is constant, n is function

❖ Lower bound

❖ Best case

---

# Ω-Omega notation

- ❖ For a given function  $g(n)$  denote by  $\Omega(g(n))$  the set of functions

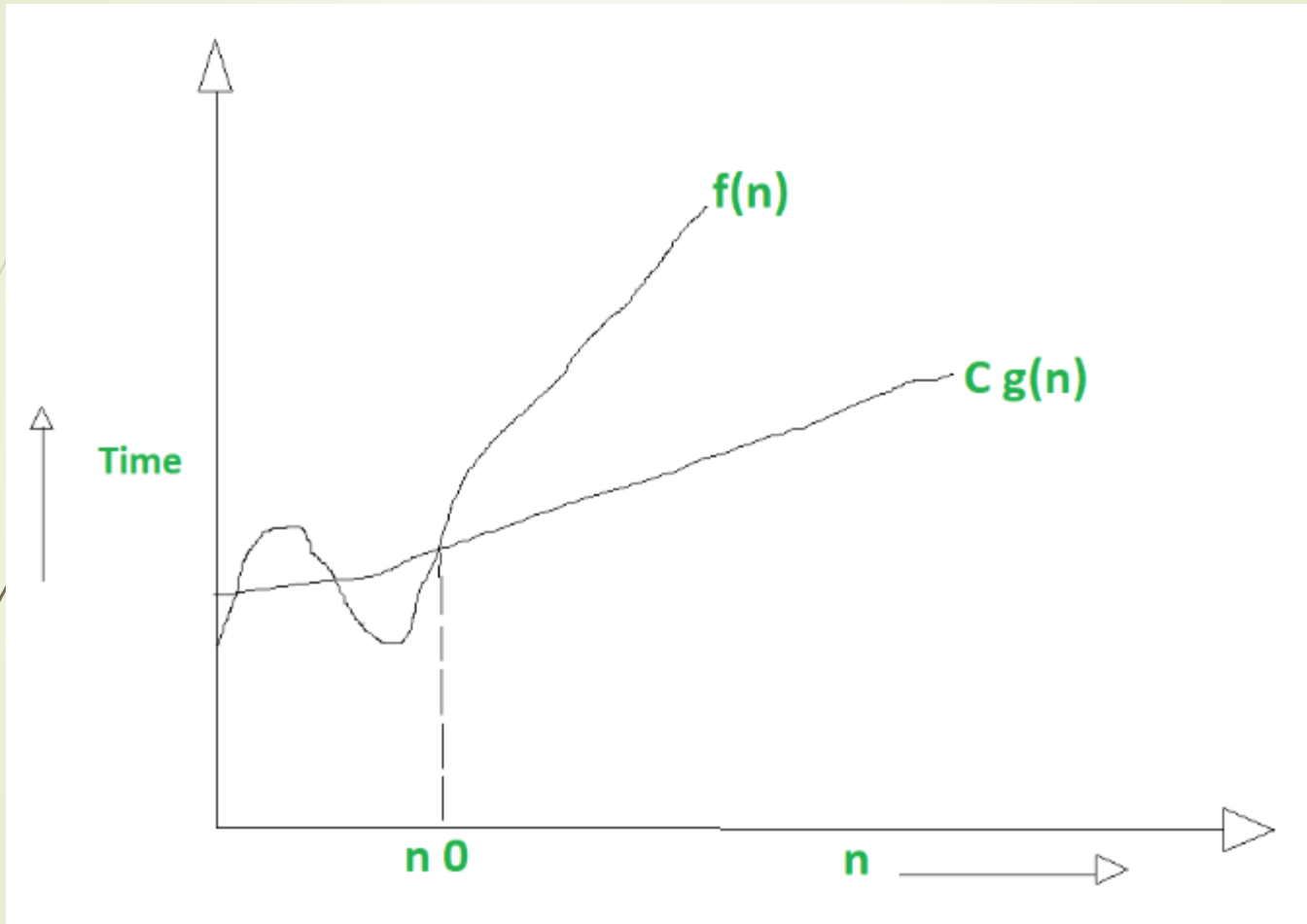
$$\Omega(g(n)) = \left\{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \right.$$

- ❖ We use  $\Omega$ -notation to give us an asymptotic lower bound on a function, to within a constant factor.

- ❖  $f(n) = \Omega(g(n))$  means that there exists some constant  $c$  s.t.  $f(n)$  is always  $\geq cg(n)$  for large enough  $n$ .

$$f(n) = \Omega(g(n))$$

$$f(n) \geq cg(n)$$



# Examples

Example :  $f(n)=3n+2$

Formula :  $f(n) \geq c g(n)$        $n \geq n_0, c > 0$   
 $, n_0 \geq 1$

$$f(n)=3n+2$$

$3n+2 \geq 1*n, c=1$  put the value of  $n=1$

$n=1$   
of  $n$

$5 \geq 1$  true     $n_0 \geq 1$  for all value

It means that  $f(n) = \Omega g(n)$ .

---

# 3. -Theta notation



Theta (Θ) notation : Asymptotic “Equality” (same rate).

It represent average bound of algorithm running time.

By using theta notation we can calculate average amount of time.

So it called average case time complexity of algorithm.

Formula :  $c_1 g(n) \leq f(n) \leq c_2 g(n)$

where c is constant, n is function

❖ Average bound

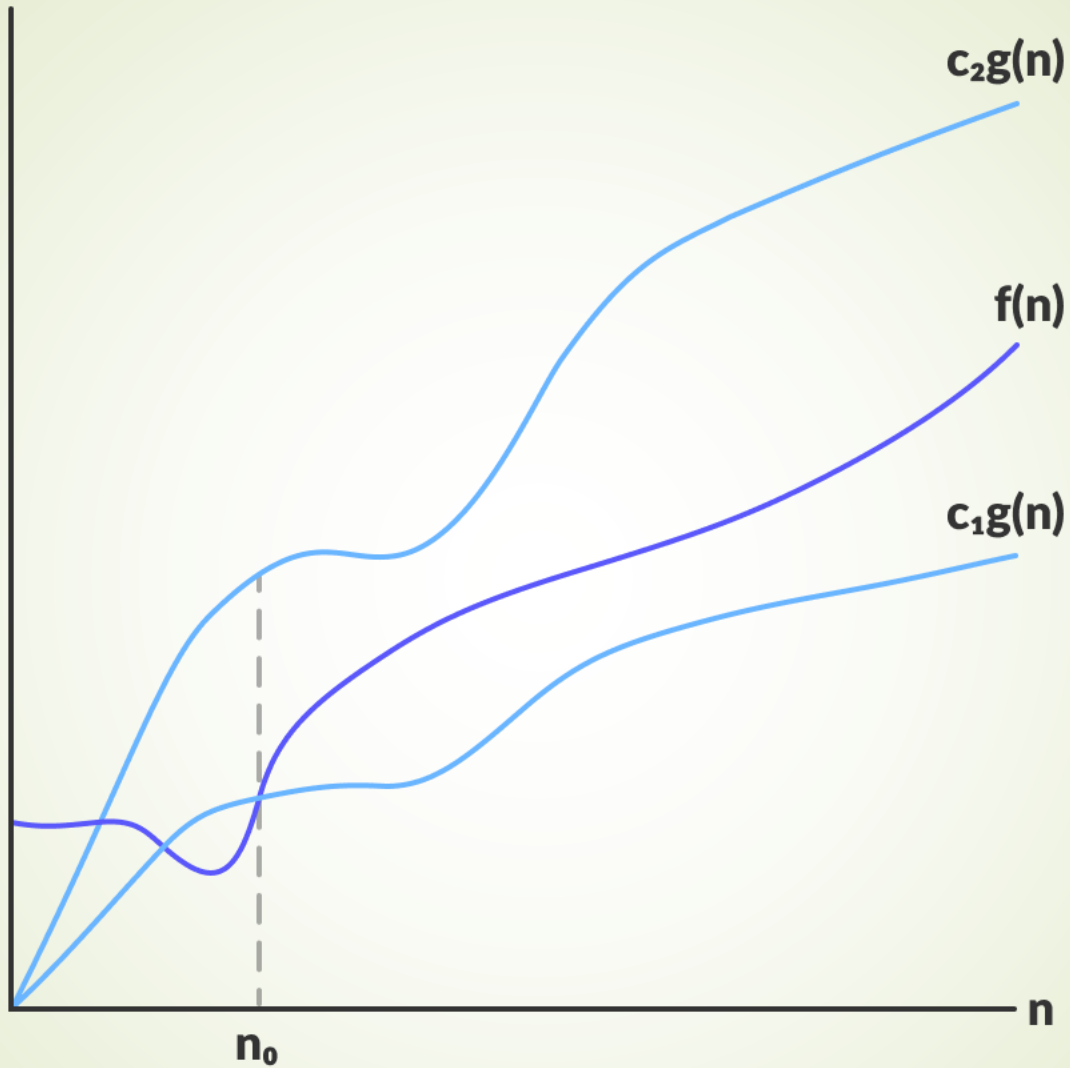
---

# -Theta notation

- ❖ For a given function  $g(n)$ , we denote by  $\Theta(g(n))$  the set of functions

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ s.t.} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- ❖ A function  $f(n)$  belongs to the set  $\Theta(g(n))$  if there exist positive constants  $c_1$  and  $c_2$  such that it can be “sandwiched” between  $c_1 g(n)$  and  $c_2 g(n)$  or sufficiently large  $n$ .
- ❖  $f(n) = \Theta(g(n))$  means that there exists some constant  $c_1$  and  $c_2$  s.t.  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for large enough  $n$ .



$$f(n) = \Theta(g(n))$$

# Examples

Example :  $f(n)=3n+2$

Formula :  $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$f(n)=2n+3$$

$1 * n \leq 3n+2 \leq 4 * n$  now put the value of  $n=1$   
we get  $1 \leq 5 \leq 4$  false

$n=2$  we get  $2 \leq 8 \leq 8$  true

$n=3$  we get  $3 \leq 11 \leq 12$  true

Now all value of  $n \geq 2$  it is true above condition is satisfied.

---




## 4. Little oh notation

- ▶ Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of  $f(n)$ .

Slower growth rate

$f(n)$  grows slower than  $g(n)$

- ▶ Let  $f(n)$  and  $g(n)$  are the functions that map positive real numbers. We can say that the function  $f(n)$  is  $o(g(n))$  if for any real positive constant  $c$ , there exists an integer constant  $n_0 \leq 1$  such that  $f(n) > 0$ .
-

- 
- ❖ Using mathematical relation, we can say that  $f(n) = o(g(n))$  means,


if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

- ❖ **Example on little o asymptotic notation:**

1. If  $f(n) = n^2$  and  $g(n) = n^3$  then check whether  $f(n) = o(g(n))$  or not.

---



Sol:

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{n^2}{n^3} \\ &= \lim_{n \rightarrow \infty} \frac{1}{n} \\ &= \frac{1}{\infty} \\ &= 0 \end{aligned}$$

The result is 0, and it satisfies the equation mentioned above. So we can say that  $f(n) = o(g(n))$ .

---

- Another asymptotic notation is little omega notation. it is denoted by  $(\omega)$ .
- Little omega  $(\omega)$  notation is used to describe a loose lower bound of  $f(n)$ .
- **Faster growth rate**
- $F(n)$  grows faster than  $g(n)$

➤ If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$  then  $f(n) = \omega(g(n))$

## Example of asymptotic notation

Problem:-Find upper bond ,lower bond & tight bond range for functions:  $f(n) = 2n+5$

Solution:-Let us given that  $f(n) = 2n+5$  , now  $g(n) = n$   
lower bond= $2n$ , upper bond = $3n$ , tight bond= $2n$

For Big -oh notation( $O$ ):- according to definition

$f(n) \leq cg(n)$  for Big oh we use upper bond so

$f(n) = 2n+5$ ,  $g(n) = n$  and  $c = 3$  according to definition

$$2n+5 \leq 3n$$

Put  $n=1$   $7 \leq 3$  false      Put  $n=2$   $9 \leq 6$  false      Put  $n=3$   $14 \leq 9$   
false      Put  $n=4$   $13 \leq 12$  false      Put  $n=5$   $15 \leq 15$  true

now for all value of  $n \geq 5$  above condition is satisfied.  $C=3$   $n \geq 5$

---

2. Big - omega notation :-  $f(n) \geq c \cdot g(n)$  we know that this

Notation is lower bound notation so  $c=2$

Let  $f(n)=2n+5$  &  $g(n)=2 \cdot n$

Now  $2n+5 \geq c \cdot g(n)$ ;

$$2n+5 \geq 2n \text{ put } n=1$$

We get  $7 \geq 2$  true for all value of  $n \geq 1, c=2$  condition is satisfied.

3. Theta notation :- according to definition

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g$$

---

## ANALYSIS OF INSERTION-SORT(CONTD.)

- The worst case: The array is reverse sorted

( $t_j = j$  for  $j=2,3, \dots,n$ ).

$$\sum_{j=1}^n j = \frac{n(n+1)}{2}$$

$$T(n) = c_1n + c_2(n-1) + c_5(n(n+1)/2 - 1)$$

$$+ c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$$

$$= (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n$$

$$T(n) = an^2 + bn + c$$

# RANDOMIZED ALGORITHMS

- ▶ A **randomized algorithm** is an **algorithm** that employs a degree of randomness as part of its logic.
  - ▶ The **algorithm** typically uses uniformly **random** bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of **random** bits.
  - ▶ An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm..
  - ▶ Example: Quick sort
-

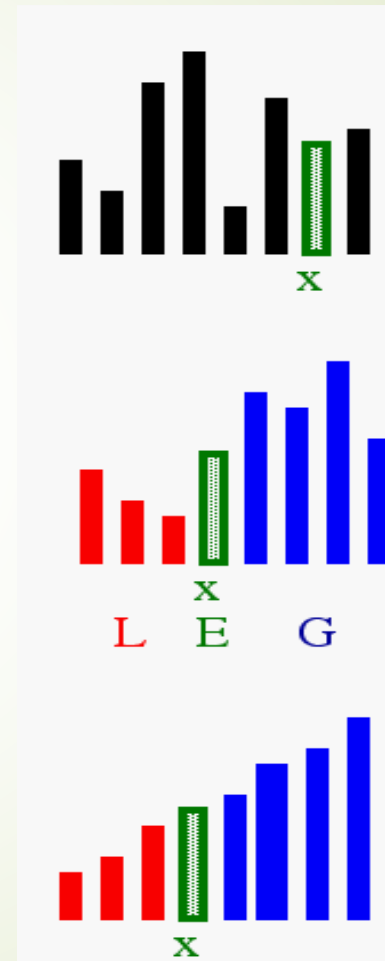


# QUICK SORT

**Select:** pick an arbitrary element  $x$  in  $S$  to be the pivot.

**Partition:** rearrange elements so that elements with value less than  $x$  go to List  $L$  to the left of  $x$  and elements with value greater than  $x$  go to the List  $R$  to the right of  $x$ .


**Recursion:** recursively sort the lists  $L$  and  $R$ .





# DIVIDE AND CONQUER

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets,  $1 < k \leq n$ , yielding 'k' sub problems.
  - These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
  - If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
-

- 
- If the problem  $p$  and the size is  $n$ , sub problems are  $n_1, n_2 \dots n_k$ , respectively, then the computing time of D And C is described by the recurrence relation.
  - $T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n); & \text{otherwise.} \end{cases}$

“Where  $T(n)$  is the time for D And C on any I/p of size  $n$ .

- $g(n)$  is the time of compute the answer directly for small I/p s.  $f(n)$  is the time for dividing P & combining the solution to sub problems.
-

# DIVIDE AND CONQUER :GENERAL METHOD

1. Algorithm D And C(P)
  2. {
  3. if small(P) then return S(P);
  4. else
  5. {
  6. divide P into smaller instances
  7.  $P_1, P_2 \dots P_k, k \geq 1$ ;
  8. Apply D And C to each of these sub problems;
  9. return combine (D And C( $P_1$ ), D And C( $P_2$ ),.....,D And C( $P_k$ ));
  10. }
  11. }
-



## EXAMPLE

Consider the case in which  $a=2$  and  $b=2$ . Let  $T(1)=2$  &  $f(n)=n$ . We have,

$$T(n) = 2T(n/2)+n$$

$$2[2T(n/2/2)+n/2]+n$$

$$[4T(n/4)+n]+n$$

$$4T(n/4)+2n$$


$$4[2T(n/4/2)+n/4]+2n$$

$$4[2T(n/8)+n/4]+2n$$

$$8T(n/8)+n+2n$$

$$8T(n/8)+3n$$

---


$$2^3T(n/2^3)+3n$$

By using substitution method

$$\text{Let } n=2^k$$

$$K=\log_2 n$$

$$K=3$$

$$2^kT(n/n)+3n$$

$$nT(1)+3N$$

$$2n+kn$$

$$2n+n\log n$$

Time complexity is  $O(n\log n)$


---



# APPLICATIONS

**1. Binary Search** is a searching algorithm. In each step, the algorithm compares the input element  $x$  with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if  $x$  is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

---



**2.Quick sort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the sub arrays on left and right of pivot element.

**3.Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

---



# BINARY SEARCH

```
1. Algorithm Bin search(a,n,x)
2. // Given an array a[1:n] of elements in non-decreasing
3. //order, n>=0,determine whether x is present and
4. // if so, return j such that x=a[j]; else return 0.
5. {
6. low:=1; high:=n;
7. while (low<=high) do
8. {
9.   mid:=[(low+high)/2];
10.  if (x<a[mid]) then high;
11.  else if(x>a[mid]) then
12.    low=mid+1;
13.  else return mid;
14. }
15. return 0; }
```

---



## EXAMPLE

1) Let us select the 14 entries.

-15,6,0,7,9,23,54,82,101,112,125,131,142,151.

Place them in  $a[1:14]$  and simulate the steps Binsearch goes through as it searches for different values of  $x$ .

Only the variables low, high & mid need to be traced as we simulate the algorithm.

We try the following values for  $x$ : 151, -14 and 9.

for 2 successful searches & 1 unsuccessful search.

---

Table. Shows the traces of Bin search on these 3 steps.

X=151

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14
		Found

x=-14

low	high	mid
1		14
1	6	3
1	2	1
2	2	2
2	1	Not found

x=9

low	high	mid
1	14	7
1		6
4		6
		5

Found

# MERGE SORT

- Another application of Divide and conquer is merge sort.
  - Given a sequence of  $n$  elements  $a[1], \dots, a[n]$  the general idea is to imagine then split into 2 sets  $a[1], \dots, a[n/2]$  and  $a[[n/2]+1], \dots, a[n]$ .
  - Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of  $n$  elements.
  - Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.
-

# ALGORITHM FOR MERGE SORT

Algorithm MergeSort(low,high)

- //a[low:high] is a global array to be sorted
  - //Small(P) is true if there is only one element
  - //to sort. In this case the list is already sorted.
  - {
  - if (low<high) then //if there are more than one element
  - {
  - //Divide P into subproblems
  - //find where to split the set
  - **mid = [(low+high)/2];**
  - //solve the subproblems.
  - mergesort (low,mid);
  - mergesort(mid+1,high); //combine the solutions .
  - merge(low,mid,high);
  - }
  - }
-

**Algorithm:** Merging 2 sorted subarrays using auxiliary storage.

```
1. Algorithm merge(low,mid,high)
2. /*a[low:high] is a global array containing two sorted subsets in a[low:mid] and in a[mid+1:high].The goal is to merge these 2 sets into a single set residing in a[low:high].b[] is an auxiliary global array.
   */
3. {
4. h=low; I=low; j=mid+1;
5. while ((h<=mid) and (j<=high)) do {
6. if (a[h]<=a[j]) then {
7. b[I]=a[h];
8. h = h+1; }
9. else {
10.b[I]= a[j];
11.j=j+1; }
12.I=I+1; }
13.if (h>mid) then
14.for k=j to high do {
15.b[I]=a[k];
16.I=I+1;
17.}
18.else
19.for k=h to mid do
20.{
21.b[I]=a[k];
22.I=I+1; }
23.for k=low to high do a[k] = b[k]; }
```

---

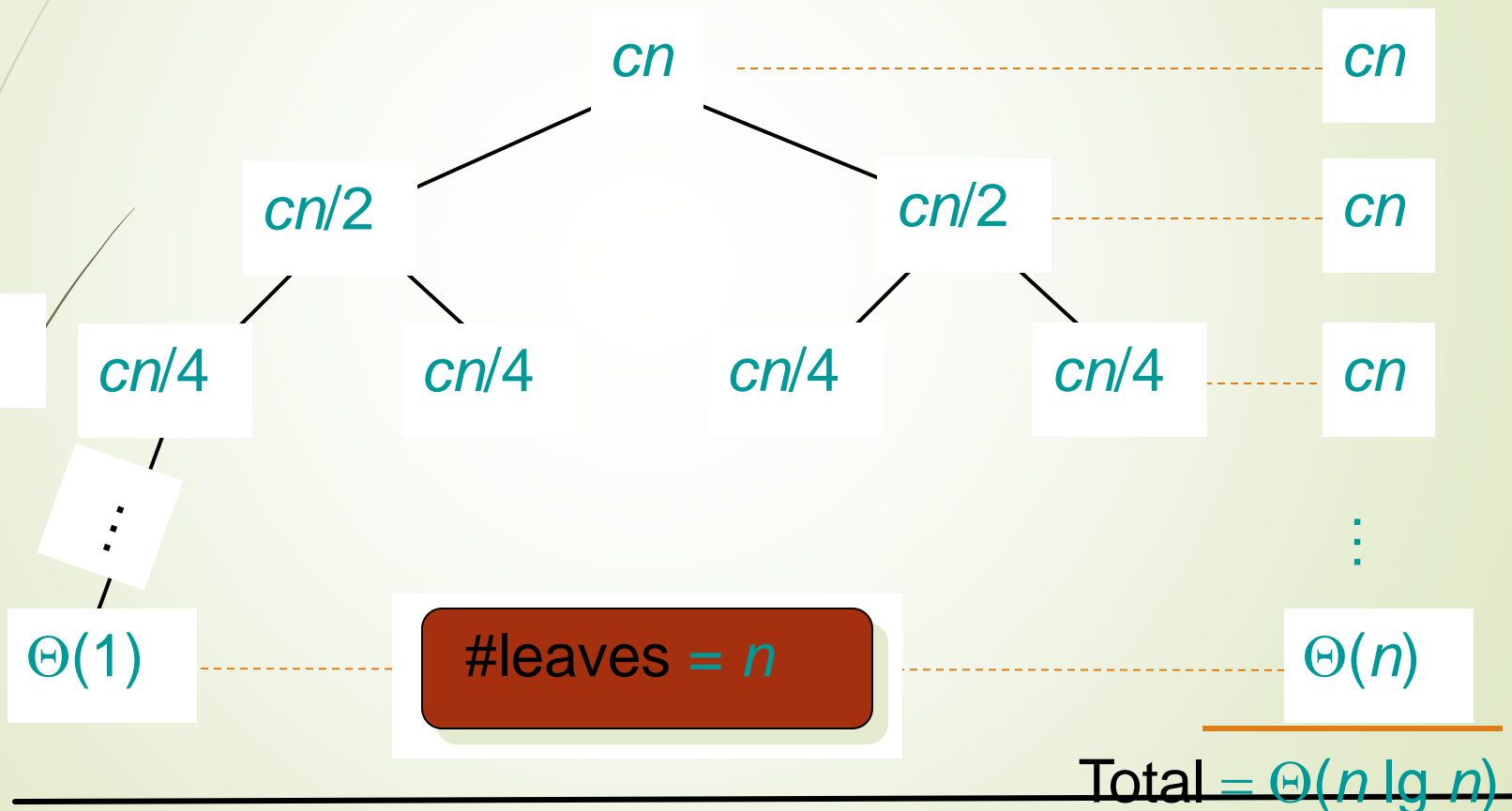
## EXAMPLE

- Consider the array of 10 elements  $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$   
Algorithm Mergesort begins by splitting  $a[]$  into 2 sub arrays each of size five ( $a[1:5]$  and  $a[6:10]$ ).
  - The elements in  $a[1:5]$  are then split into 2 sub arrays of size 3 ( $a[1:3]$ ) and 2 ( $a[4:5]$ )
  - Then the items in  $a[1:3]$  are split into sub arrays of size 2 ( $a[1:2]$ ) & one ( $a[3:3]$ )
  - The 2 values in  $a[1:2]$  are split to find time into one-element sub arrays and now the merging begins.
-

# Recursion tree

Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

$h = \lg n$





# QUICK SORT

- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
  - This is accomplished by rearranging the elements in  $a[1:n]$  such that  $a[i] \leq a[j]$  for all  $i$  between 1 &  $m$  and all  $j$  between  $(m+1)$  &  $n$  for some  $m$ ,  $1 \leq m \leq n$ .
  - Thus the elements in  $a[1:m]$  &  $a[m+1:n]$  can be independently sorted.
  - No merge is needed. This rearranging is referred to as partitioning.
-

1. **Algorithm:** Partition the array  $a[m:p-1]$  about  $a[m]$

2. Algorithm Partition( $a,m,p$ )

/\*within  $a[m],a[m+1],\dots,a[p-1]$  the elements are rearranged in such a manner that if initially  $t=a[m]$ ,then after completion  $a[q]=t$  for some  $q$  between  $m$  and

$p-1, a[k]\leq t$  for  $m\leq k<q$ , and  $a[k]>t$  for  $q<k<p$ .  $q$  is returned Set  $a[p]=\text{infinite}$ . \*/

3. {

4.  $v=a[m];l=m;j=p;$

5. repeat

6. {

7. repeat

8.  $l=l+1;$

9. until( $a[l]>=v$ );

10. repeat

11.  $j=j-1;$

12. until( $a[j]\leq v$ );

13. if ( $l<j$ ) then interchange( $a,l,j$ );

14. }until( $l>=j$ );

15.  $a[m]=a[j]; a[j]=v;$

16. return  $j$ ;

17. }

18. Algorithm Interchange( $a,l,j$ ) //Exchange  $a[l]$  with  $a[j]$

19. {

20.  $p=a[l];$

21.  $a[l]=a[j];$

22.  $a[j]=p;$

23. }

## ➤ **Algorithm:** Sorting by Partitioning

Algorithm Quicksort(p,q)

//Sort the elements  $a[p], \dots, a[q]$  which resides

- //is the global array  $a[1:n]$  into ascending  
//order;  $a[n+1]$  is considered to be defined
  - // and must be  $\geq$  all the elements in  $a[1:n]$
  - {
  - if( $p < q$ ) then // If there are more than one element
  - {
  - // divide p into 2 subproblems
  - $j = \text{partition}(a, p, q+1);$
  - //“j” is the position of the partitioning element.
  - //solve the subproblems.
  - quicksort(p, j-1);
  - quicksort(j+1, q);
  - //There is no need for combining solution.
  - }
  - }
-



# **Graph Coloring using Backtracking**

---



# Backtracking

- ▶ In many real world problems, a solution can be obtained by exhaustively searching through a large but finite number of possibilities. Hence, the need arose for developing systematic techniques of searching, with the hope of cutting down the search space to possibly a much smaller space.
  - ▶ Here, we present a general technique for organizing the search known as **backtracking**. This algorithm design technique can be described as an organized exhaustive search which often avoids searching all possibilities.
-

# The 3-Coloring Problem

- Given an undirected graph  $G=(V, E)$ , it is required to color each vertex in  $V$  with one of three colors, say 1, 2, and 3, such that no two adjacent vertices have the same color. We call such a coloring **legal**; otherwise, if two adjacent vertices have the same color, it is **illegal**.
  - A coloring can be represented by an  $n$ -tuple  $(c_1, c_2, \dots, c_n)$  such that  $c_i \in \{1, 2, 3\}$ ,  $1 \leq i \leq n$ .
  - For example,  $(1, 2, 2, 3, 1)$  denotes a coloring of a graph with five vertices.
-

# The 3-Coloring Problem

- There are  $3^n$  possible colorings (legal and illegal) to color a graph with  $n$  vertices.
  - The set of all possible colorings can be represented by a complete ternary tree called the **search tree**. In this tree, each path from the root to a leaf node represents one coloring assignment.
  - An incomplete coloring of a graph is **partial** if no two adjacent colored vertices have the same color.
  - Backtracking works by generating the underlying tree one node at a time.
  - If the path from the root to the current node corresponds to a legal coloring, the process is terminated (unless more than one coloring is desired).
-

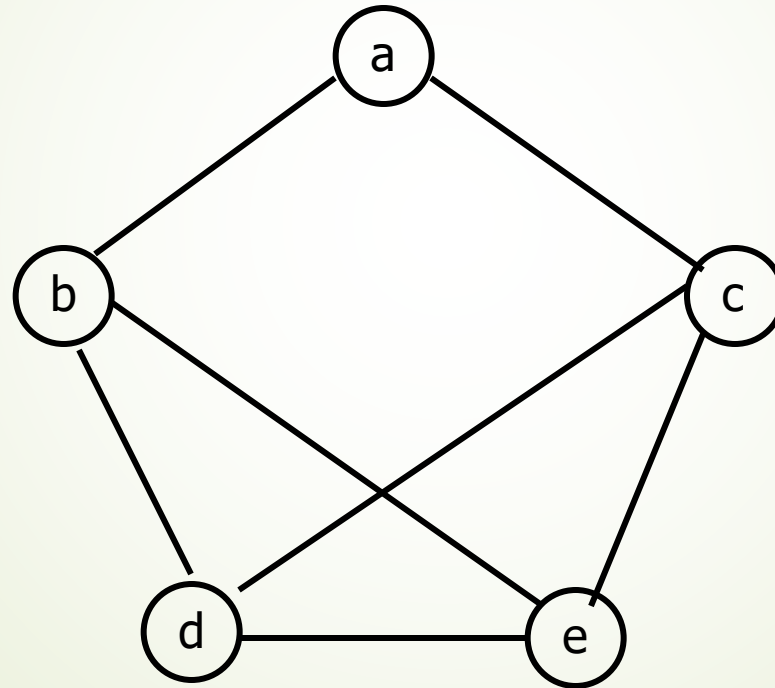
# The 3-Coloring Problem

- If the length of this path is less than  $n$  and the corresponding coloring is partial, then one child of the current node is generated and is marked as the current node.
  - If, on the other hand, the corresponding path is not partial, then the current node is marked as a **dead node** and a new node corresponding to another color is generated.
  - If, however, all three colors have been tried with no success, the search backtracks to the parent node whose color is changed, and so on.
-



# The 3-Coloring Problem

➤ Example:





# The 3-Coloring Problem

There are two important observations to be noted, which generalize to all backtracking algorithms:

(1) The nodes are generated in a depth-first-search manner.

(2) There is no need to store the whole search tree; we only need to store the path from the root to the current active node. In fact, no physical nodes are generated at all; the whole tree is implicit. We only need to keep track of the color assignment.

---

# The 3-Coloring Problem

## Recursive Algorithm

**Input:** An undirected graph  $G=(V, E)$ .

**Output:** A 3-coloring  $c[1..n]$  of the vertices of  $G$ , where each  $c[j]$  is 1, 2, or 3.

1. for  $k \leftarrow 1$  to  $n$
2.  $c[k] \leftarrow 0$ ;
3. end for;
4.  $flag \leftarrow false$ ;
5.  $graphcolor(1)$ ;
6. if  $flag$  then output  $c$ ;
7. else output "no solution";

$graphcolor(k)$

1. for  $color=1$  to 3
  2.  $c[k] \leftarrow color$ ;
  3. if  $c$  is a legal coloring then set  $flag \leftarrow true$  and exit;
  4. else if  $c$  is partial then  $graphcolor(k+1)$ ;
  5. end for;
-

# The 3-Coloring Problem

## Iterative Algorithm

**Input:** An undirected graph  $G=(V, E)$ .

**Output:** A 3-coloring  $c[1 \dots n]$  of the vertices of  $G$ , where each  $c[j]$  is 1, 2, or 3.

```
1. for  $k \leftarrow 1$  to  $n$ 
2.    $c[k] \leftarrow 0$ ;
3. end for;
4.  $flag \leftarrow false$ ;
5.  $k \leftarrow 1$ ;
6. while  $k \geq 1$ 
7.   while  $c[k] \leq 2$ 
8.      $c[k] \leftarrow c[k] + 1$ ;
9.     if  $c$  is a legal coloring then set  $flag \leftarrow true$  and exit from the two while loops;
10.    else if  $c$  is partial then  $k \leftarrow k + 1$ ;
11.   end while;
12.    $c[k] \leftarrow 0$ ;
13.    $k \leftarrow k - 1$ ;
14. end while;
15. if  $flag$  then output  $c$ ;
16. else output "no solution";
```

---

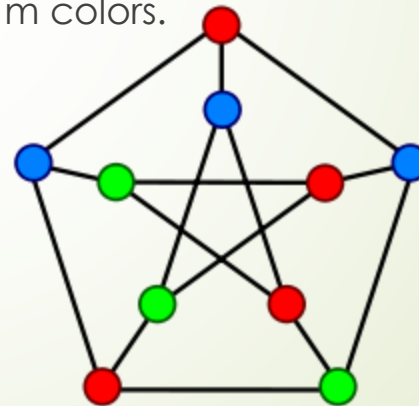
Given an undirected graph and a number  $m$ , determine if the graph can be colored with at most  $m$  colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.


➤ *Input:*

- 1) A 2D array  $graph[V][V]$  where  $V$  is the number of vertices in graph and  $graph[V][V]$  is adjacency matrix representation of the graph. A value  $graph[i][j]$  is 1 if there is a direct edge from  $i$  to  $j$ , otherwise  $graph[i][j]$  is 0.
- 2) An integer  $m$  which is maximum number of colors that can be used.

➤ *Output:*


An array  $color[V]$  that should have numbers from 1 to  $m$ .  $color[i]$  should represent the color assigned to the  $i$ th vertex. The code should also return false if the graph cannot be colored with  $m$  colors.





```
#include<stdio.h>
int G[50][50],x[50]; //G:adjacency matrix,x:colors
void next_color(int k){
    int i,j;
    x[k]=1; //coloring vertex with color1
    for(i=0;i<k;i++)
    { //checking all k-1 vertices-backtracking
      if(G[i][k]!=0 && x[k]==x[i]) //if connected and has same color
        x[k]=x[i]+1; //assign higher color than x[i]
    }
}
```





```
int main(){
    int n,e,i,j,k,l;
    printf("Enter no. of vertices : ");
    scanf("%d",&n); //total vertices
    printf("Enter no. of edges : ");
    scanf("%d",&e); //total edges

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            G[i][j]=0; //assign 0 to all index of adjacency matrix

    printf("Enter indexes where value is 1-->\n");
    for(i=0;i<e;i++){
        scanf("%d %d",&k,&l);
        G[k][l]=1;
        G[l][k]=1;
    }
}
```

---



```
➤ for(i=0;i<n;i++)  
    next_color(i); //coloring each vertex
```

```
➤  
printf("Colors of vertices -->\n");  
for(i=0;i<n;i++) //displaying color of each vertex  
    printf("Vertex[%d] : %d\n",i+1,x[i]);
```

```
return 0;  
}
```

---





# Union-Find structure

---

# Basic set operations

*find(a)*

*union(S<sub>1</sub>, S<sub>2</sub>, S)*

*member(a, S)*

*add(a, S)*

*remove(a, S)*

*min(S)*

Given several sets. Find the one,

where  $a$  belongs.  $S := S_1 \cup S_2$        $S_1$        $S_2$

Form union of sets  $S_1 \cap S_2 = \emptyset$ .

Usually supposed that  $a \in S$ .

Does element  $a$  belong to set  $S$ ?  $S := S \cup \{a\}$

Add element  $a$  to set  $S$ .

Remove element from set  $S$  if it is in that set.

Suppose that set  $S$  is linearly ordered.

Find the smallest element of set  $S$ .



# Union-Find structure

➤ An abstract data type

type `set(T)` has

procedure `createset(x: T)` returns `set`

procedure `findset(x: T)` returns `set`

procedure `union(S1,S2: set)` returns `set`

---

▶ `createset(x)` forms a set consisting of one element  $\{x\}$

`findset(x)` returns the set where  $x$  belongs

`union(S1,S2)` forms the union of the sets  $S1$  and  $S2$ .

- ▶ In union-operation the sets  $S1$  and  $S2$  are destroyed. So no element can belong to more than one set.
  - ▶ We are interested in a task, which consists of a sequence of operations `createset`, `union` and `findset`.
-



# Trivial solution

Representing a set by a list

$A \cup B$

- ▶ can be formed in constant time by combining the lists
  - ▶ findset  $O(n)$ , when there are  $n$  elements
-



# Trivial solution

Representing a set by a bit vector

- ▶ Let  $U$  be an ordered base set and  $|U| = n$
- ▶ Representing subset  $S \subseteq U$  as an  $n$ -bit vector  
    's  $i$ :th bit is 1, if  $U$ 's  $i$ :th element belongs to  $S$ . ✓
- ▶ Union can be implemented as bit vector operations (in one step, if  $n$  is not too big); requires time  $O(|U|)$  and each set requires space  $O(|U|)$ .
- ▶ Findset requires time  $O(n)$ .

$v_s : v_s$



# Trivial solution

Representing a set as a table

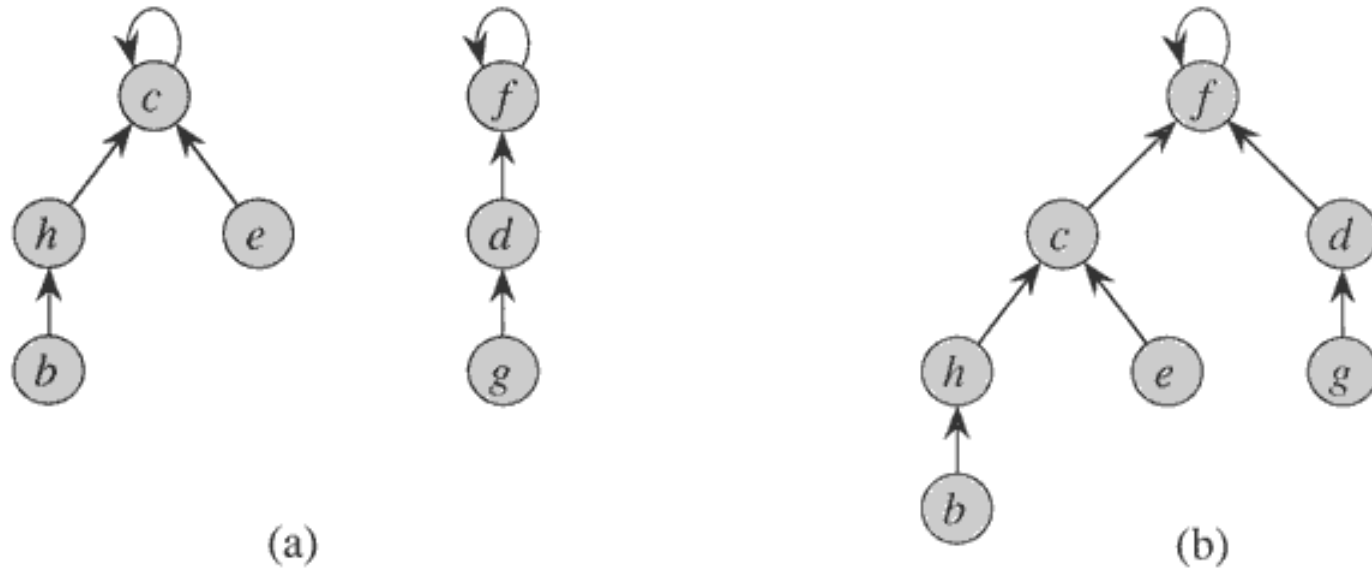
- ▶ union requires time  $O(n)$
  - ▶ findset can be implemented in constant time, if elements have order, otherwise  $O(n)$
-





# Tree representation

- ▶ Sets are represented by a forest (a single set is represented as a tree)
  - ▶ We choose the root node of a tree to be the representative of the set
  - ▶ if vertex  $x$  is the root of the tree  $T$ , then by notation  $[x]$  we mean the set formed by the vertices of the tree  $T$ .
-



**Figure 21.4** A disjoint-set forest. (a) Two trees representing the two sets of Figure 21.2. The tree on the left represents the set  $\{b, c, e, h\}$ , with  $c$  as the representative, and the tree on the right represents the set  $\{d, f, g\}$ , with  $f$  as the representative. (b) The result of  $\text{UNION}(e, g)$ .



# Tree implementation

- Operation  $\text{makeset}(x)$  forms a tree, the only vertex will be the root  $x$
  - In operation  $\text{findset}(x)$  a path is followed from vertex  $x$  upwards until the root  $y$  is reached. Then  $[y]$  is the result.
  - Operation  $\text{union}([x],[y])$  is implemented by setting vertex  $x$  as a child of vertex  $y$ . Then  $[y]$  is the union set.
  - Problem: the tree may come inbalanced
-

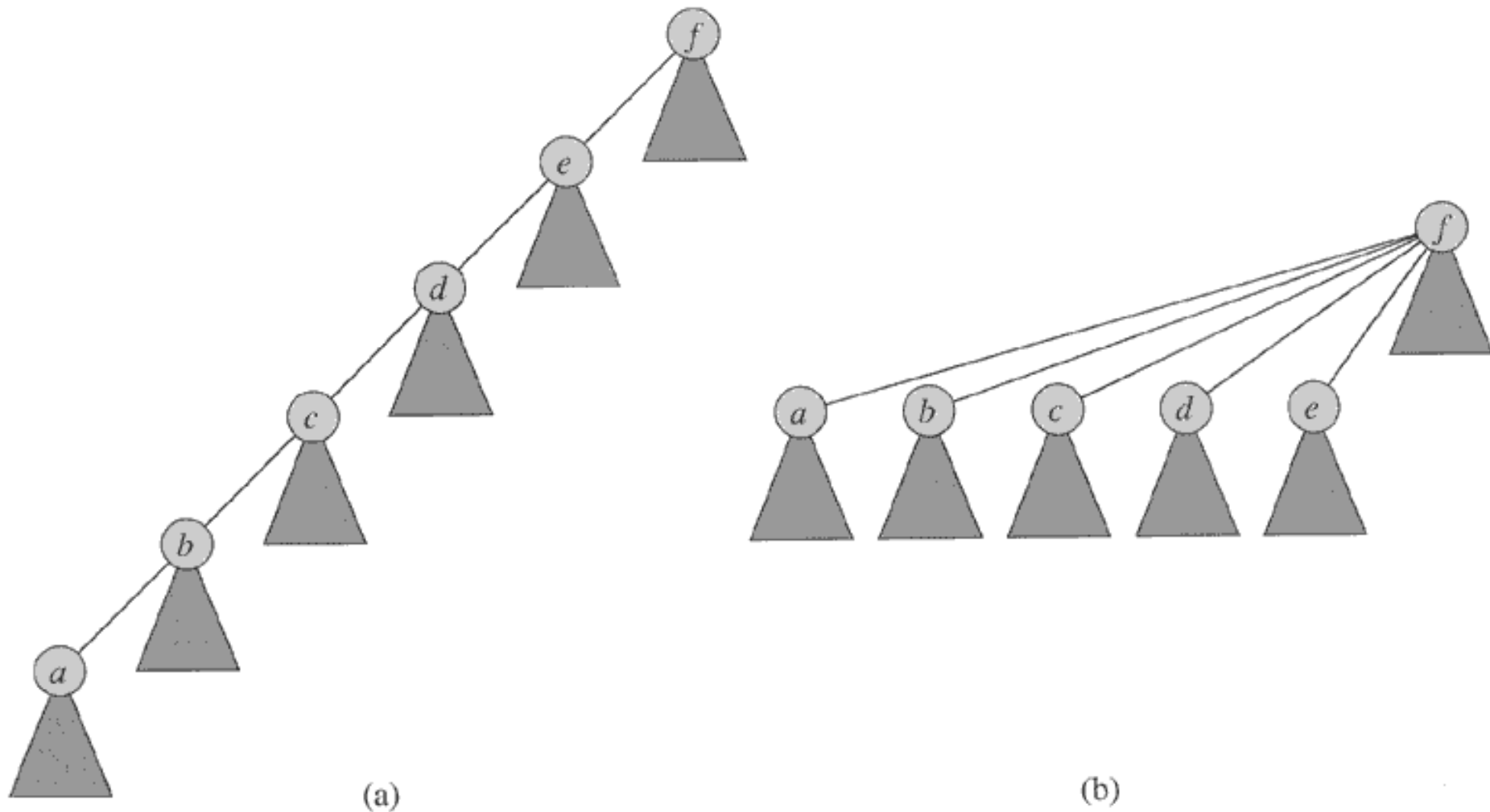


# Solutions to inbalanced trees

Solution 1: Balancing. In operation  $\text{union}([x],[y])$  the new root will be that element  $x$  or  $y$ , of which tree is highest.

Solution 2: Path compression. When a root  $y$  has been found as a result of operation  $\text{findset}(x)$ , the father of all the vertices in the path leading from  $x$  to  $y$  will be set  $y$ .

---



**Figure 21.5** Path compression during the operation  $\text{FIND-SET}$ . Arrows and self-loops at roots are omitted. (a) A tree representing a set prior to executing  $\text{FIND-SET}(a)$ . Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. (b) The same set after executing  $\text{FIND-SET}(a)$ . Each node on the find path now points directly to the root.



# Time complexity

- ▶ We examine an operation sequence, where there are  $n$  makeset-operations and  $m$  findset-operations
  - ▶ New elements are created only with makeset operation, so  $n$  is the number of elements and  $n-1$  is an upper bound for union-operations.
  - ▶ In spite of balancing, a tree may be formed, of which height is  $\log n$ . If we estimate all find-operations this difficult, the whole task would require time  $O(m \log n)$ . This estimate is too pessimistic.
-

# Time complexity

- ▶ A more accurate analysis is based on the idea of balancing the costs
- ▶ Let  $A$  be Ackerman function and  $\alpha(m, n)$  its one kind of inverse function
- ▶  $\alpha(m, n)$  grows extremely slowly.  $\alpha(m, n) \leq 3$  with all thinkable values of arguments  $m$  and  $n$ .
- ▶ If union-find task has  $n$  union- and  $m$  findset-operations, it can be executed in time  $O(n + m \times \alpha(m, n))$ . (proof omitted).

---

$$O(n + m \times \alpha(m, n))$$



# Applications



---





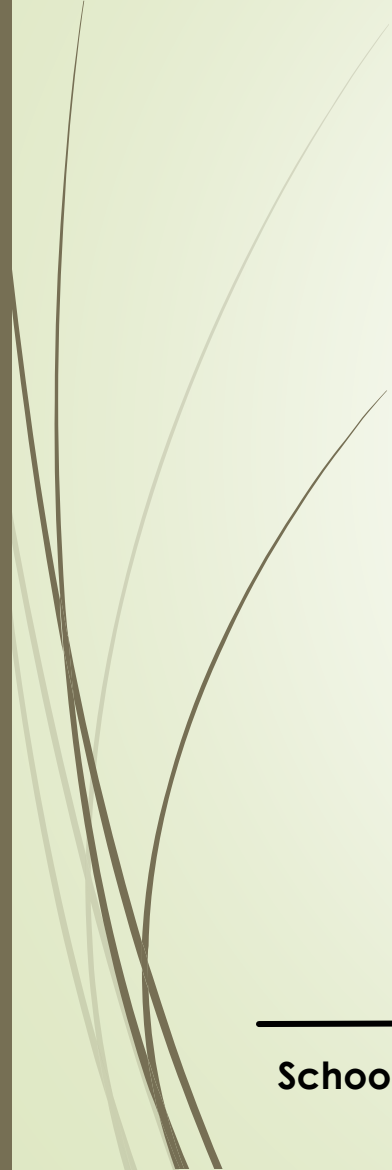
## CONNECTED-COMPONENTS ( $G$ )

```
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( $u, v$ )
```

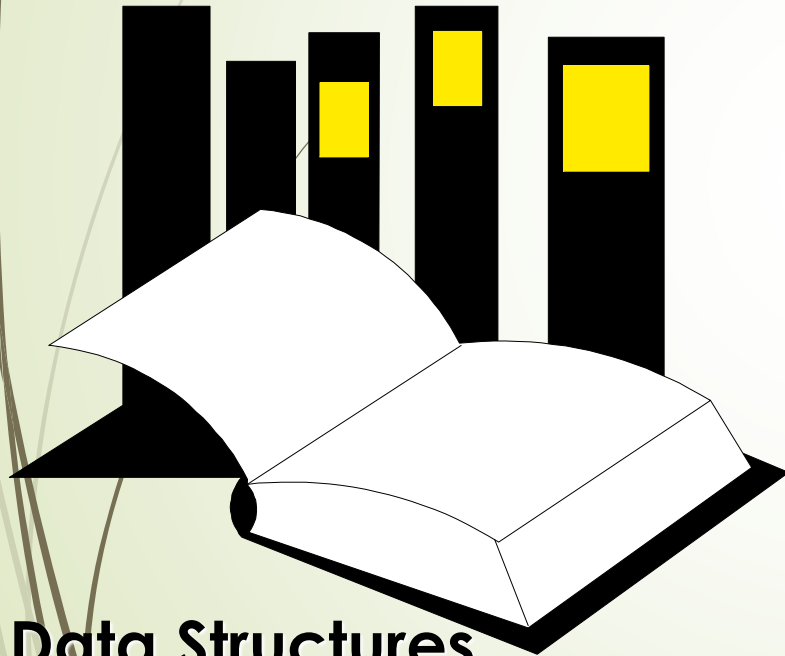
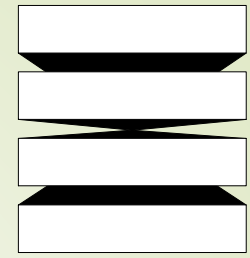


# SAME-COMPONENT( $u, v$ )

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



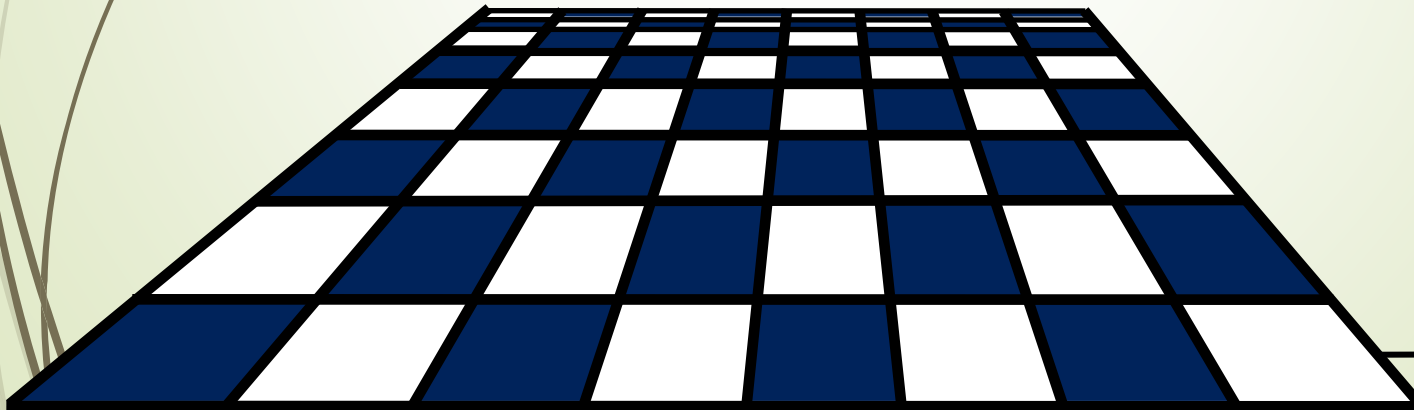
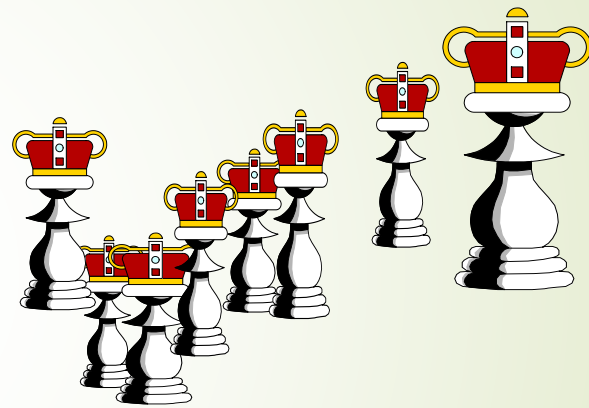
# Using a Stack



- This presentation shows another use called backtracking to solve the N-Queens problem.

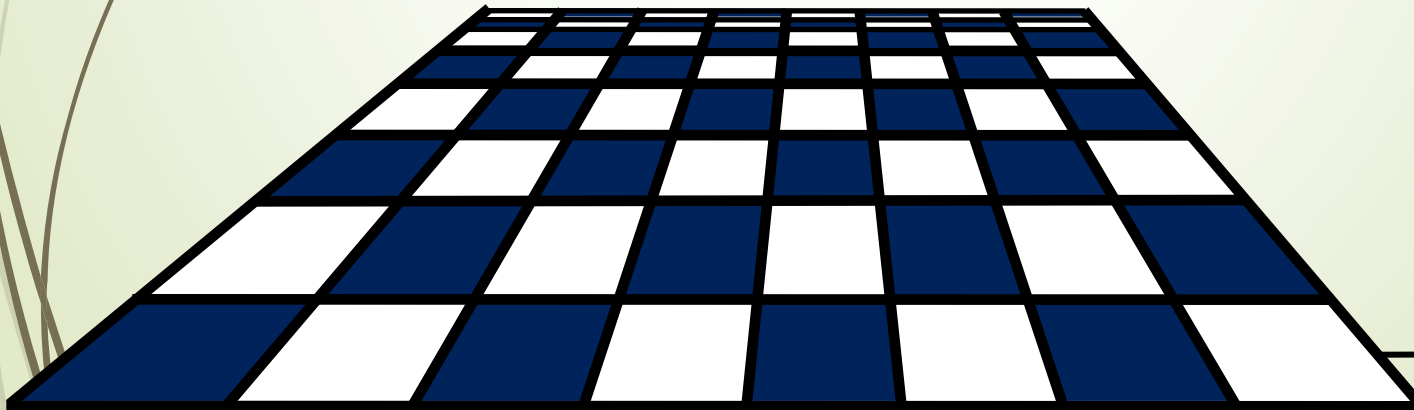
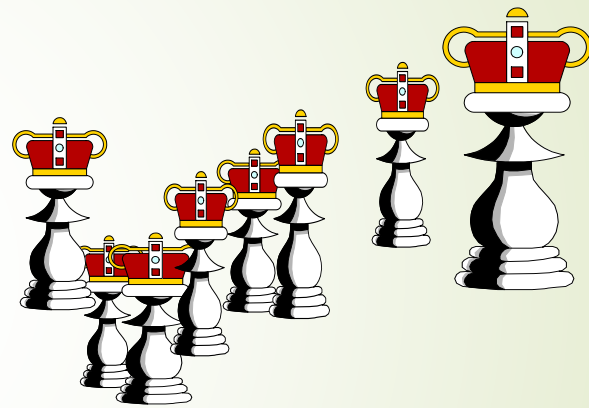
# The N-Queens Problem

- Suppose you have 8 chess queens...
- ...and a chess board



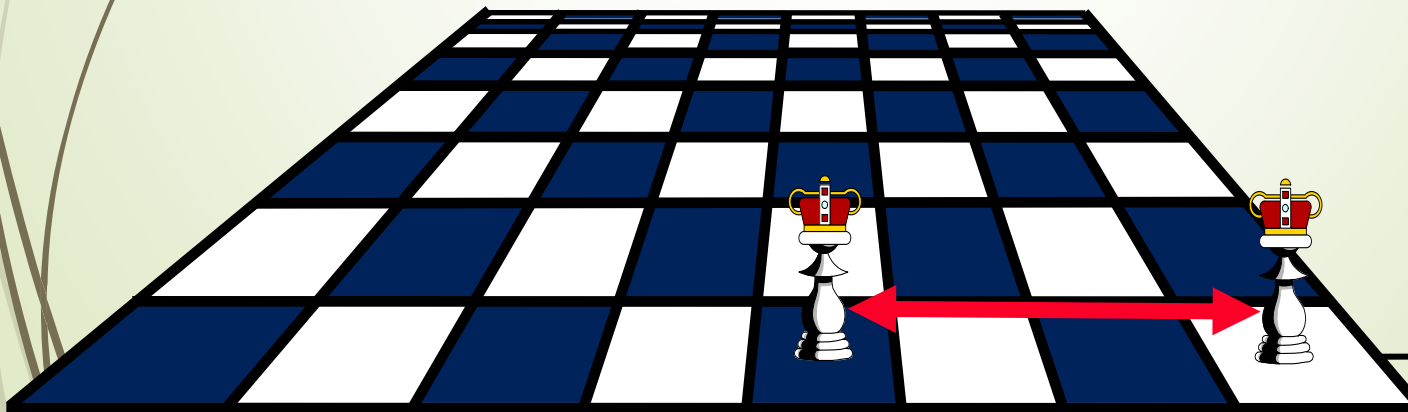
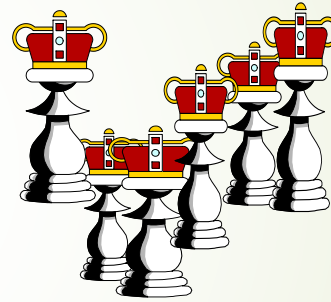
# The N-Queens Problem

Can the queens be placed on the board so that no two queens are attacking each other



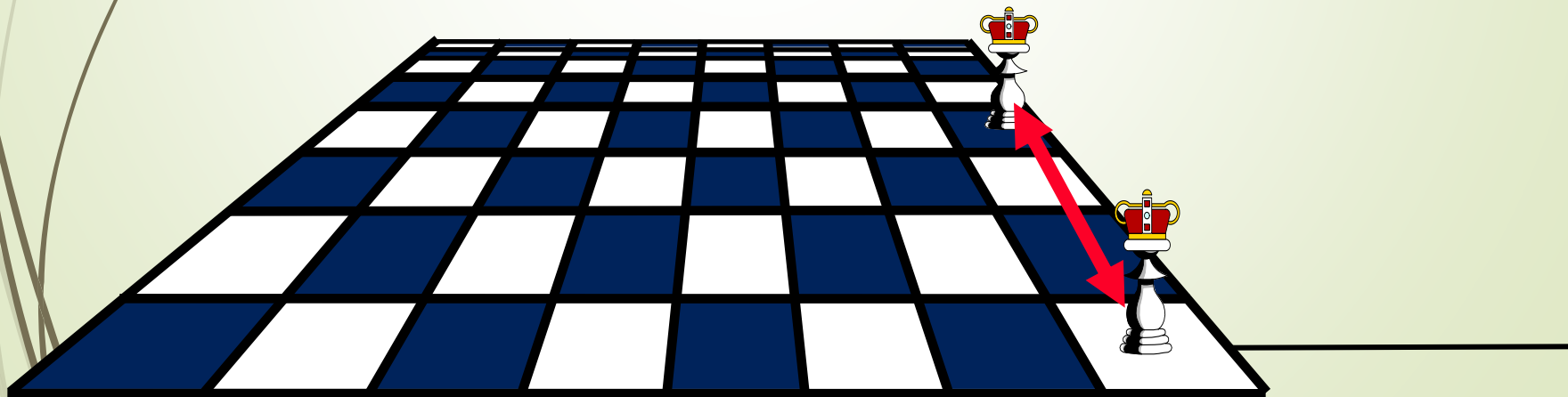
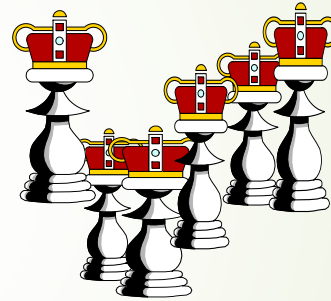
# The N-Queens Problem

Two queens are not allowed in the same row...



# The N-Queens Problem

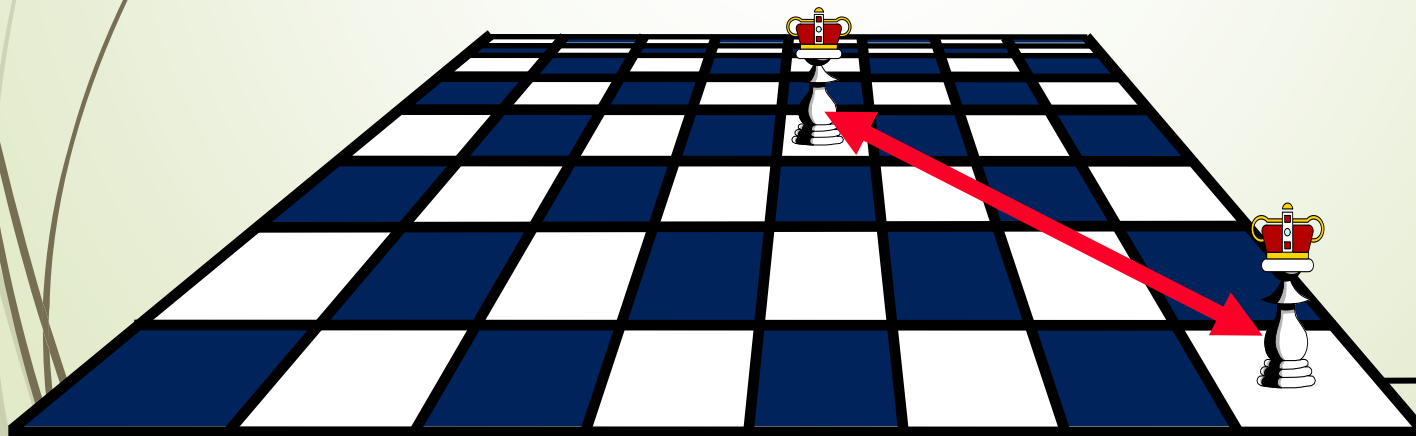
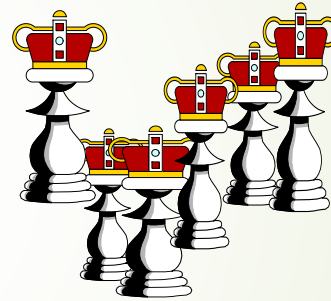
Two queens are not allowed in the same row, or in the same column...





# The N-Queens Problem

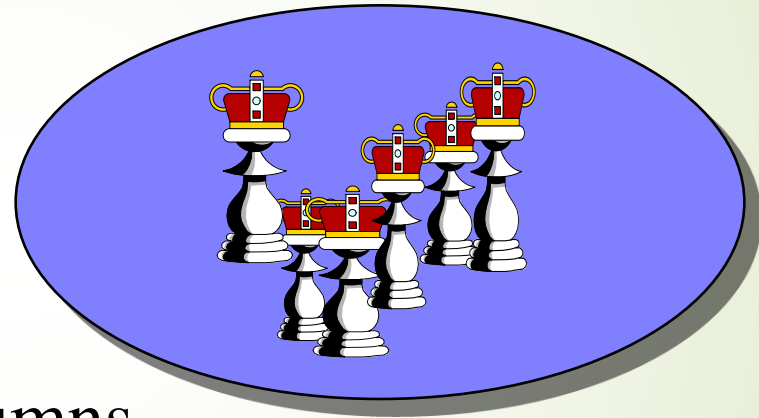
Two queens are not allowed in the same row, or in the same column, or along the same diagonal.



# The N-Queens Problem

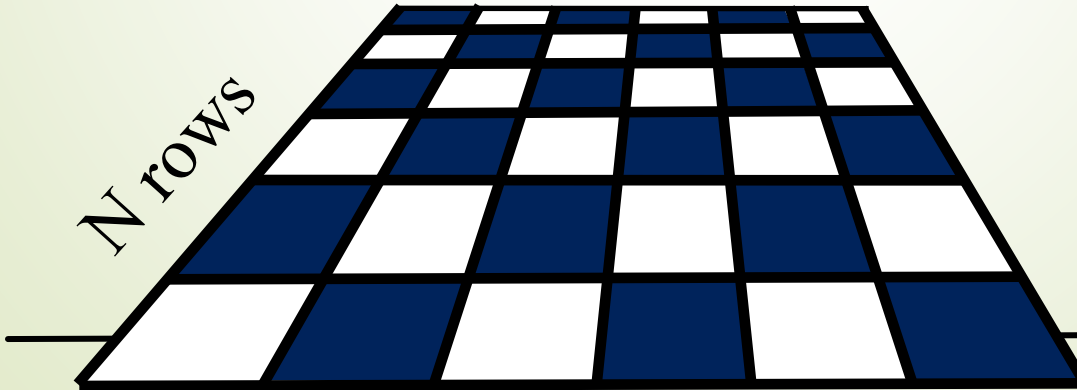
The number of queens, and the size of the board can vary.

N Queens



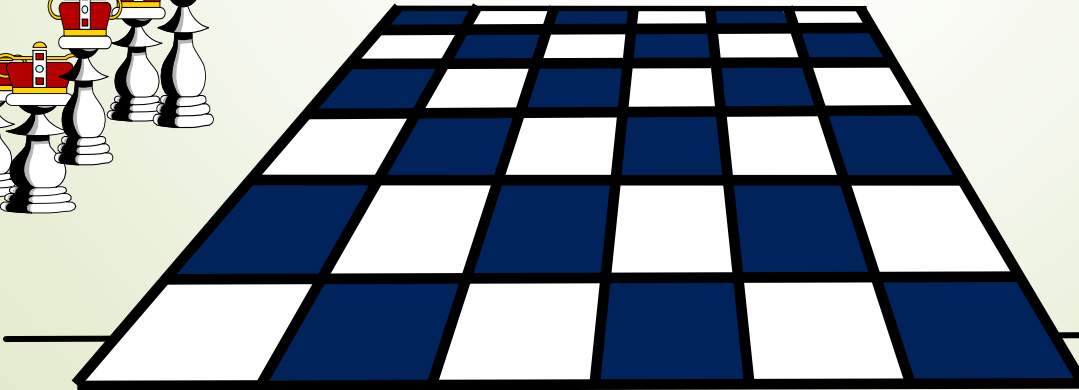
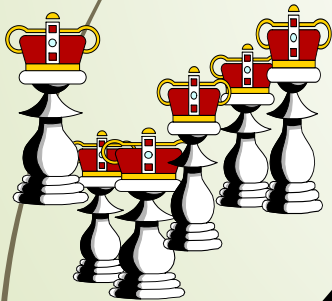
N columns

N rows



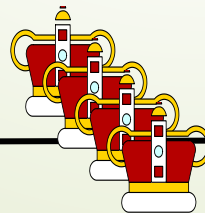
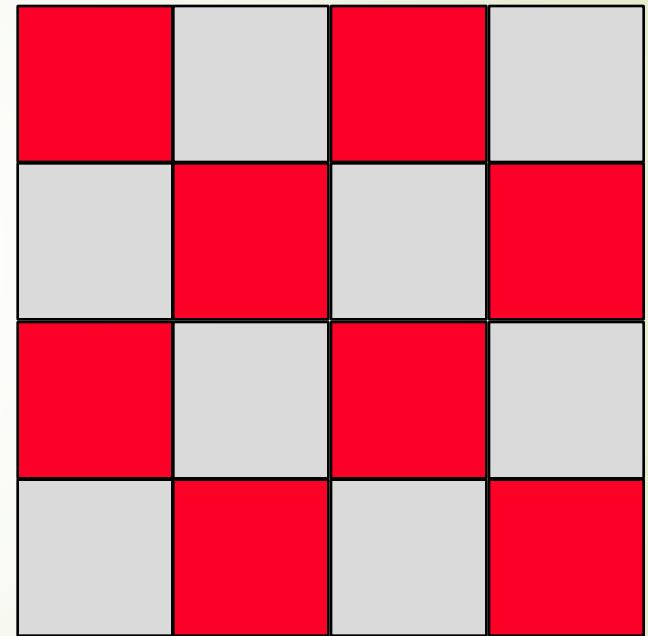
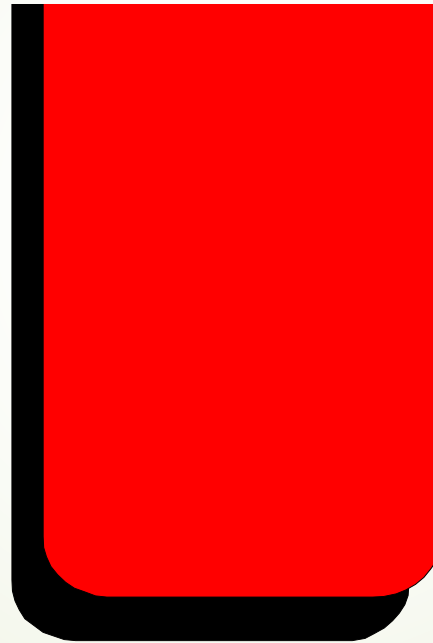
# The N-Queens Problem

We will write a program which tries to find a way to place  $N$  queens on an  $N \times N$  chess board.



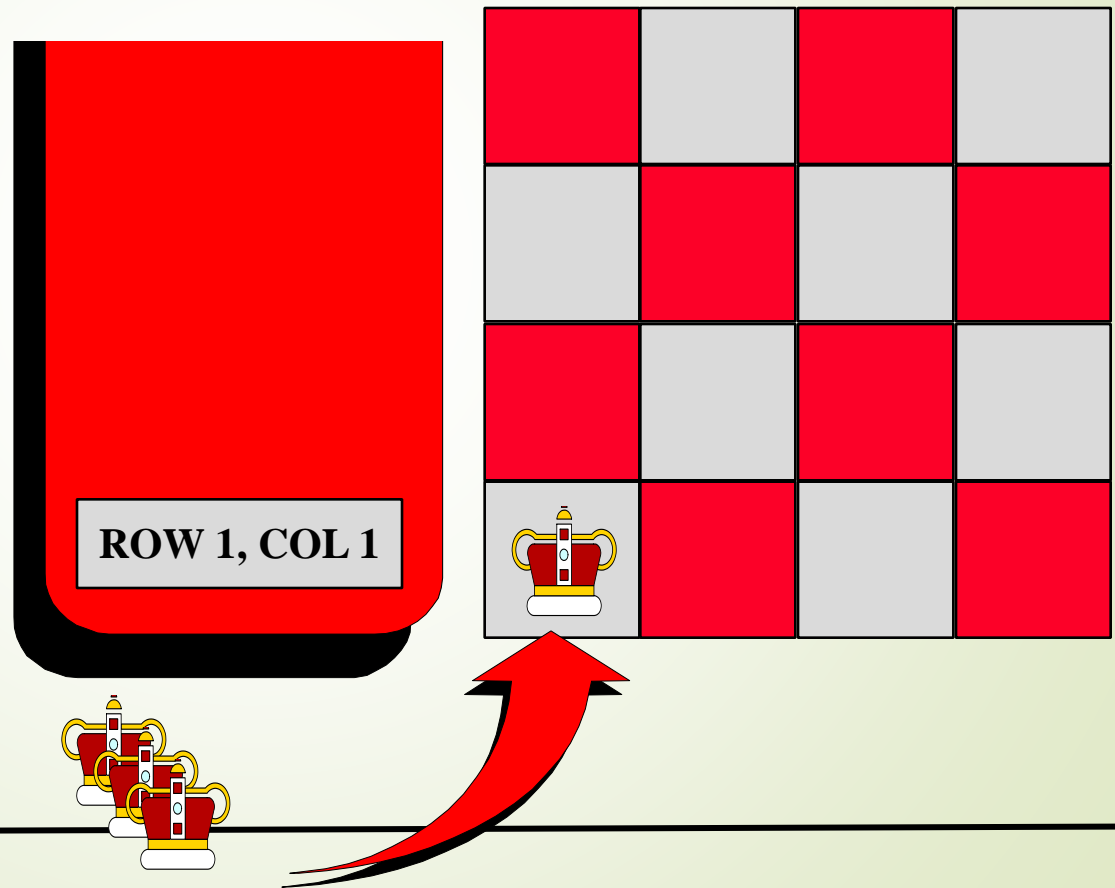
# How the program works

The program uses a stack to keep track of where each queen is placed.



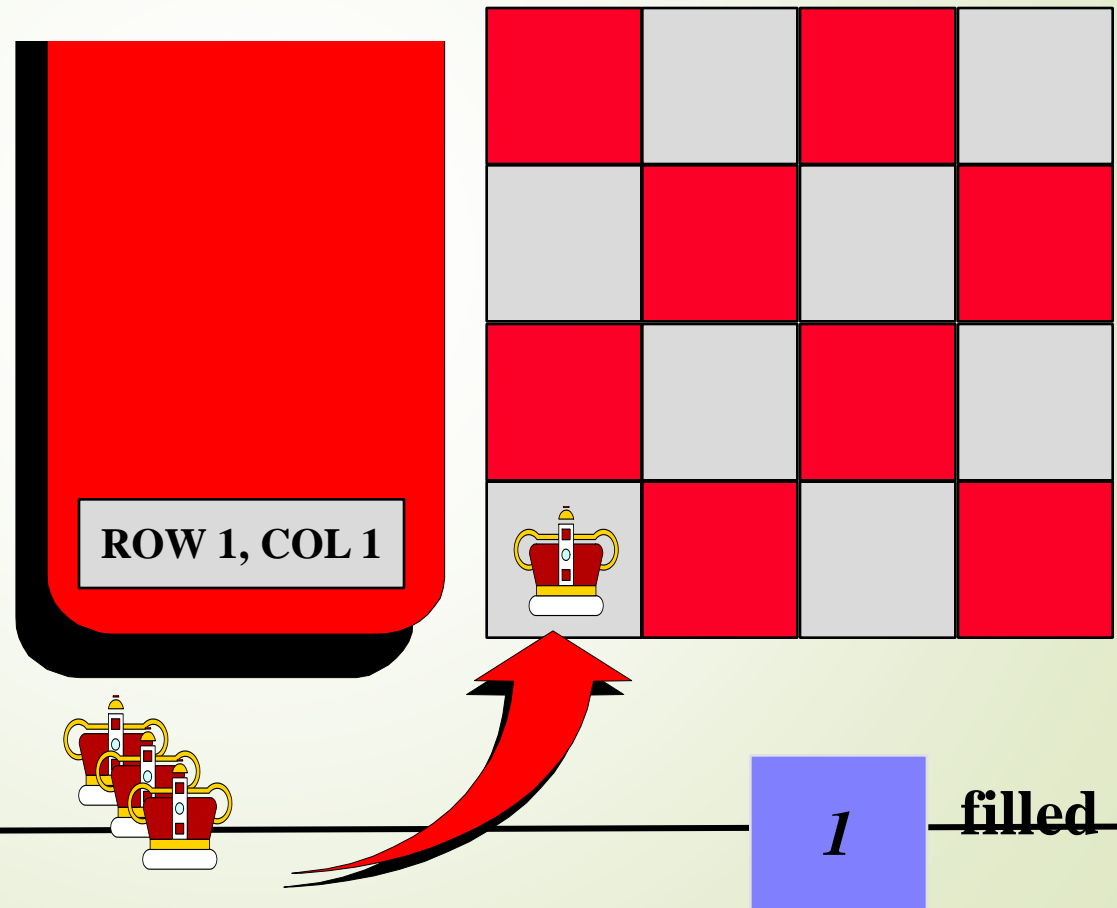
# How the program works

Each time the program decides to place a queen on the board, the position of the new queen is stored in a record which is placed in the stack.



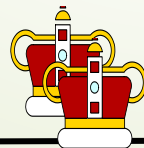
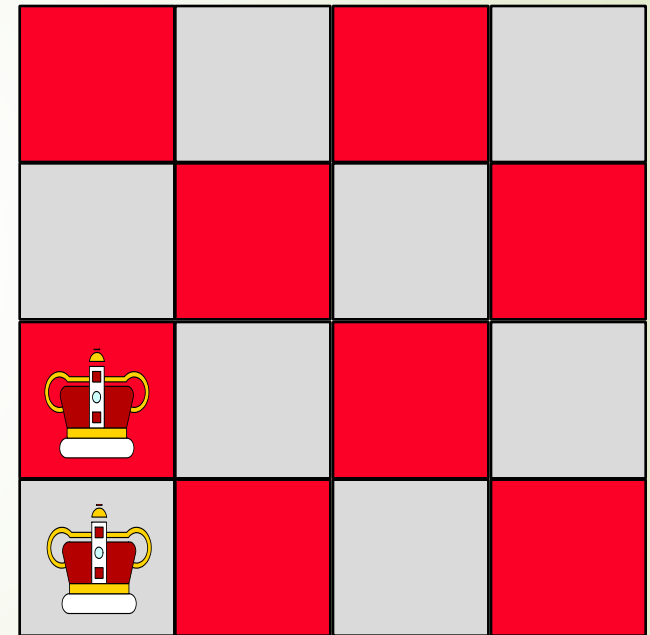
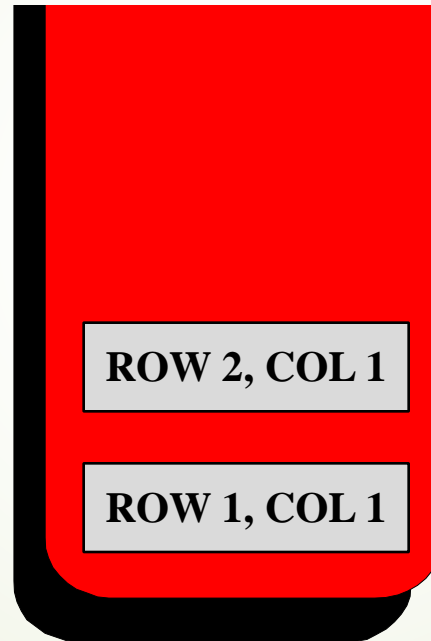
# How the program works

We also have an integer variable to keep track of how many rows have been filled so far.



# How the program works

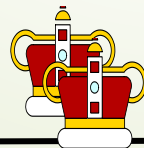
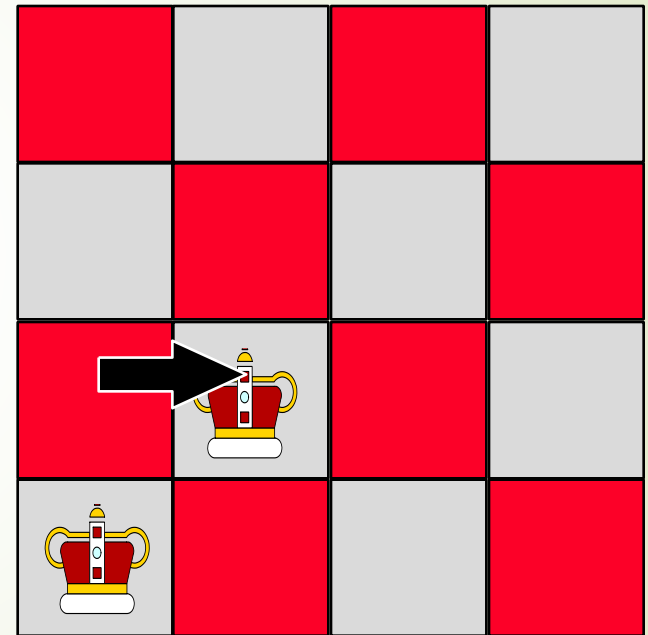
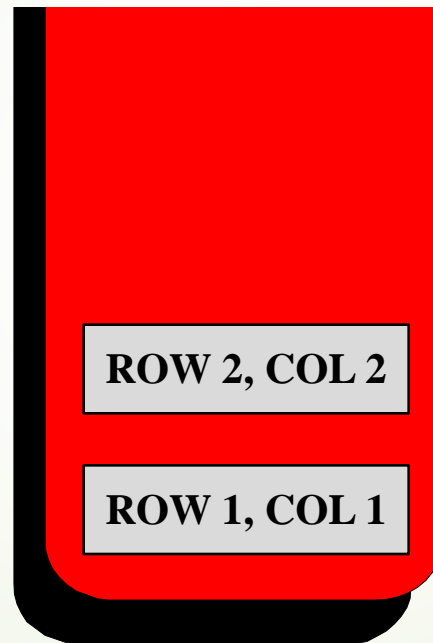
Each time we try to place a new queen in the next row, we start by placing the queen in the first column...



filled

# How the program works

...if there is a conflict with another queen, then we shift the new queen to the next column.

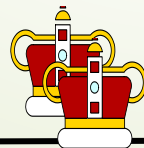
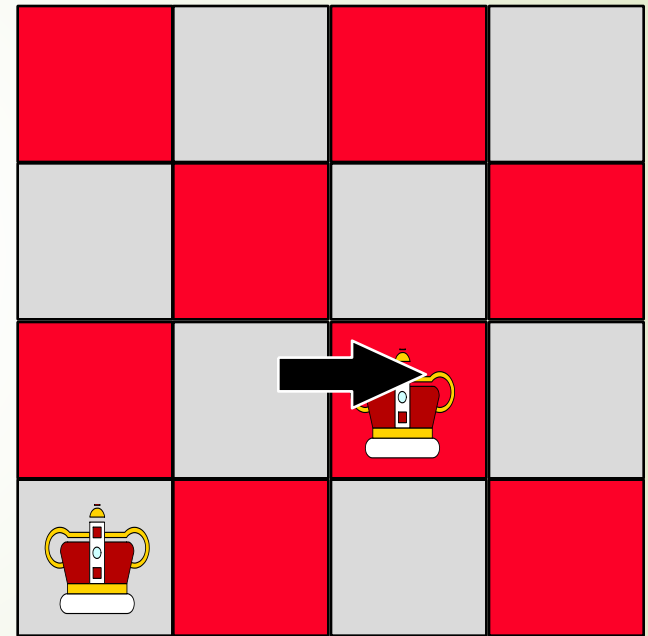
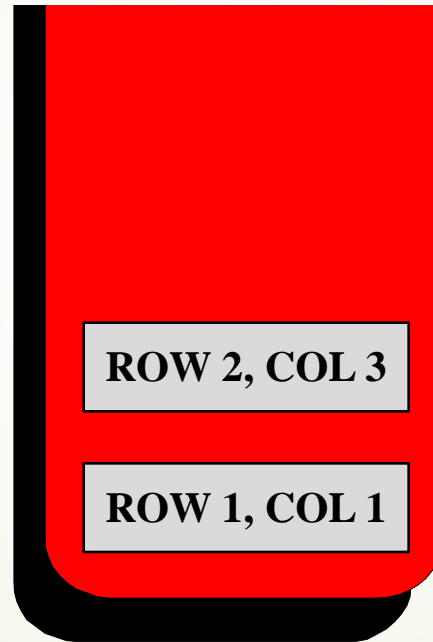


filled



# How the program works

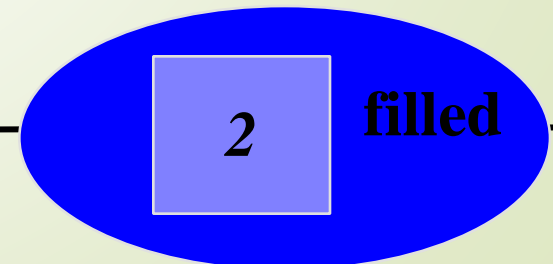
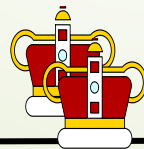
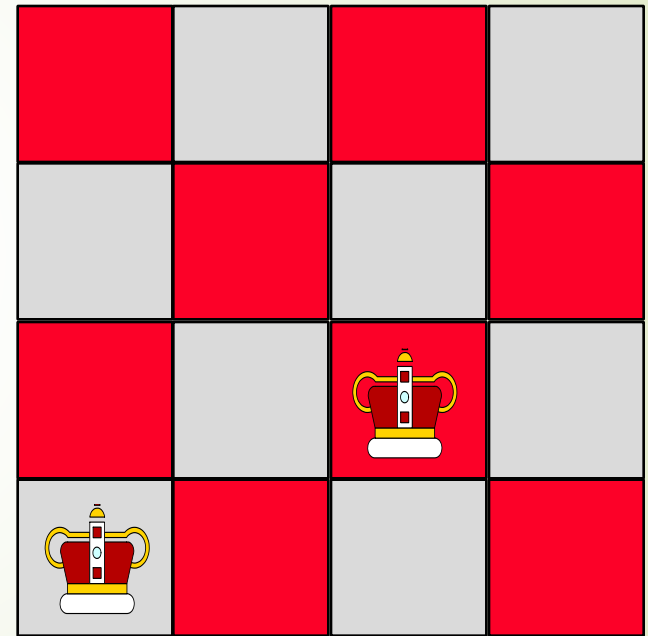
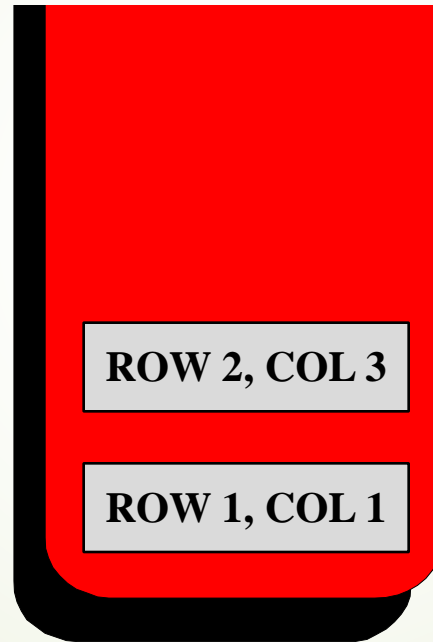
If another conflict occurs, the queen is shifted rightward again.



filled

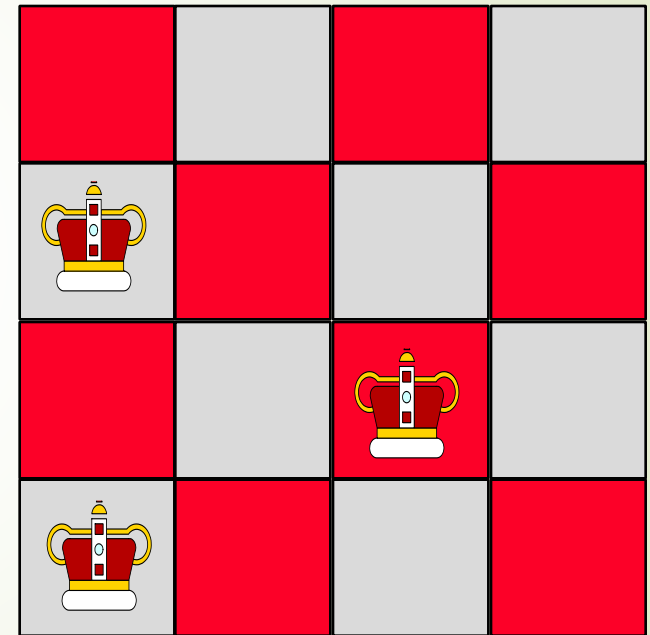
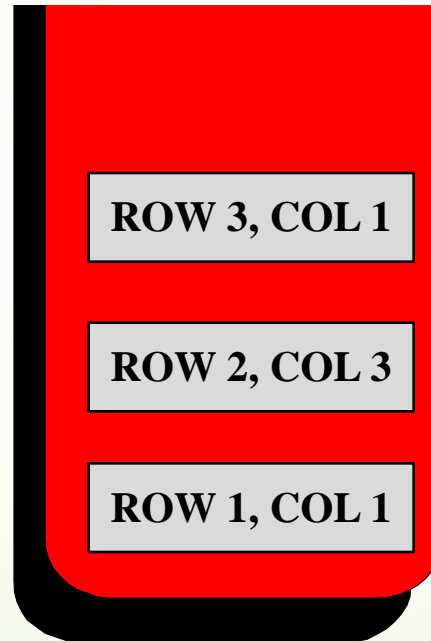
# How the program works

When there are no conflicts, we stop and add one to the value of filled.



# How the program works

Let's look at the third row. The first position we try has a conflict...

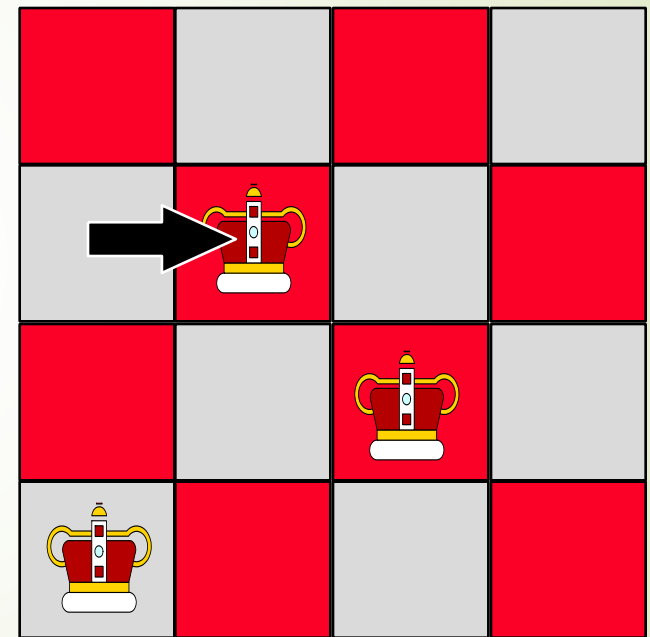
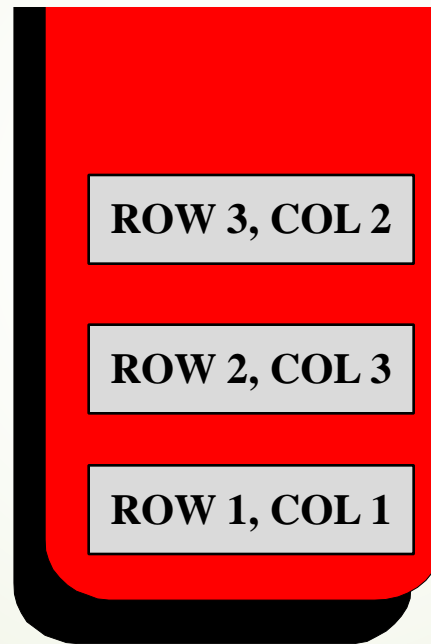


2

filled

# How the program works

...so we shift to column 2. But another conflict arises...



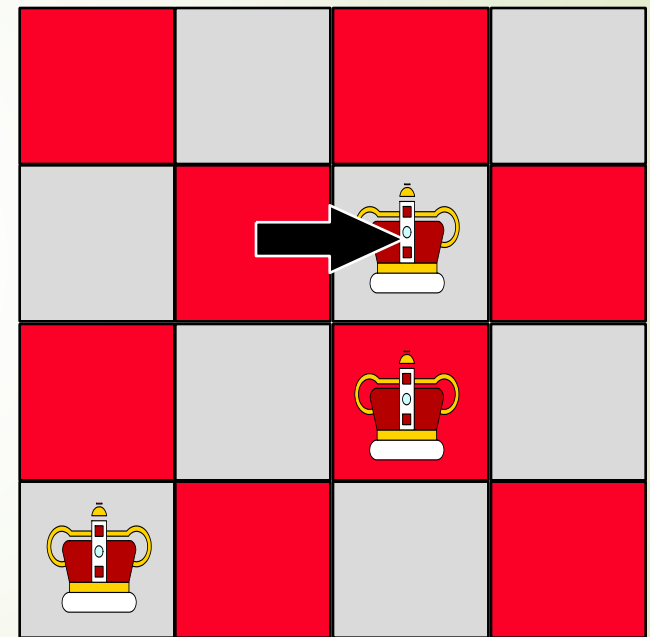
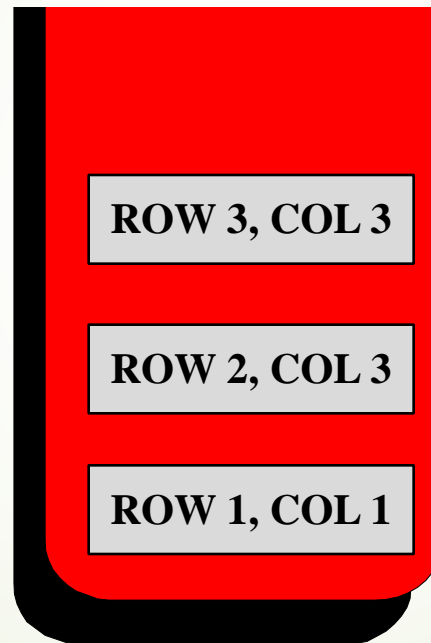
2

filled

# How the program works

...and we shift to the third column.

Yet another conflict arises...

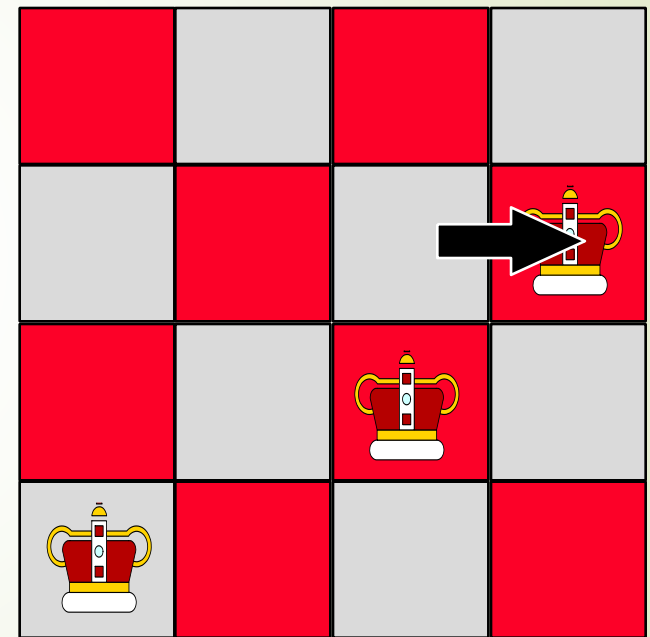
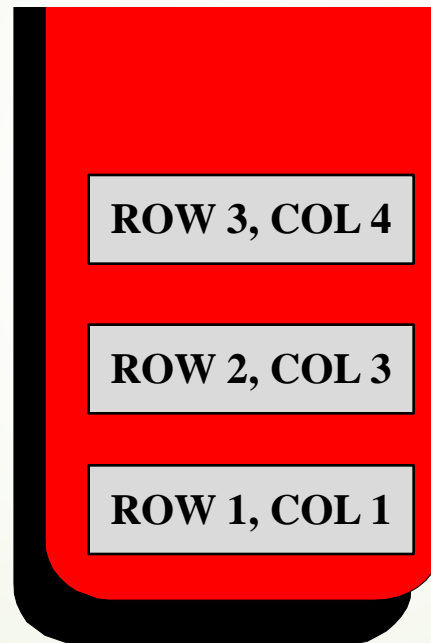


2

filled

# How the program works

...and we shift to column 4. There's still a conflict in column 4, so we try to shift rightward again...

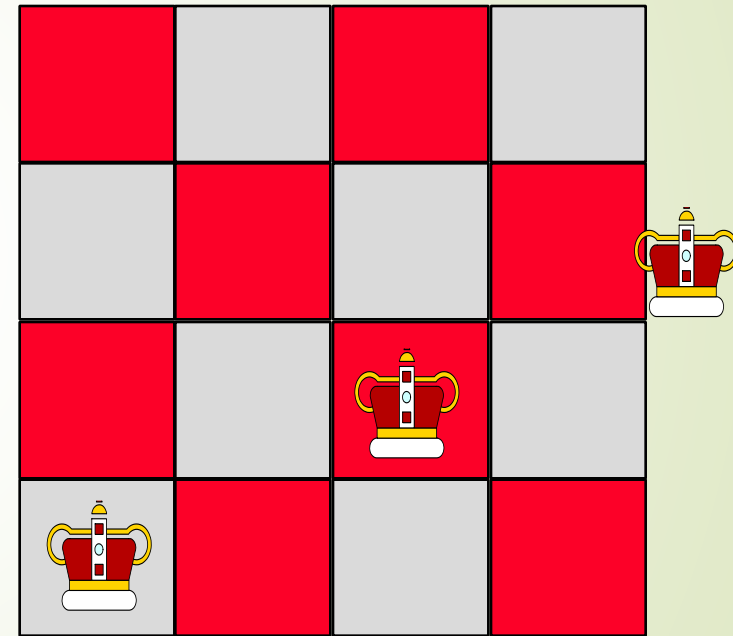
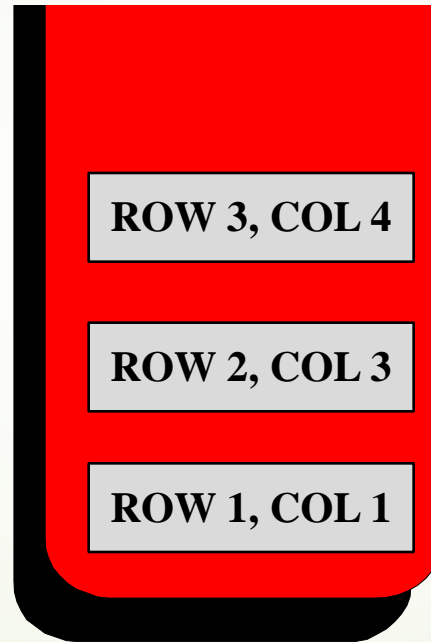


2

filled

# How the program works

...but there's nowhere else to go.



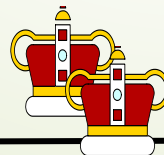
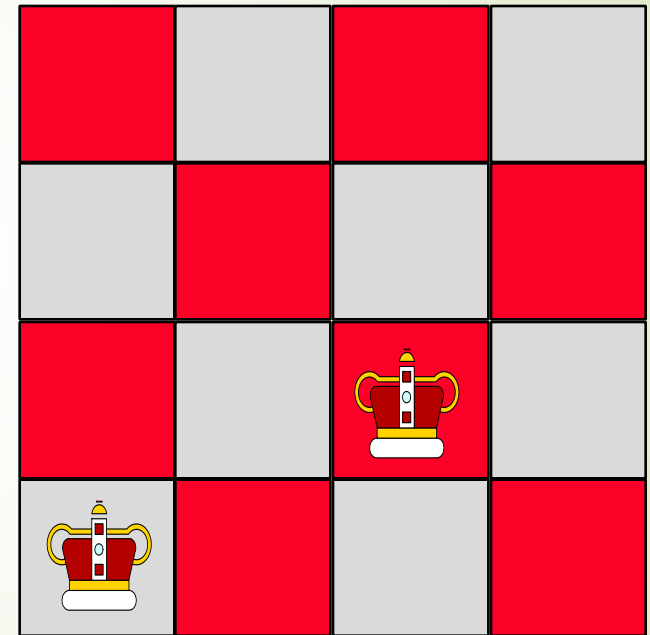
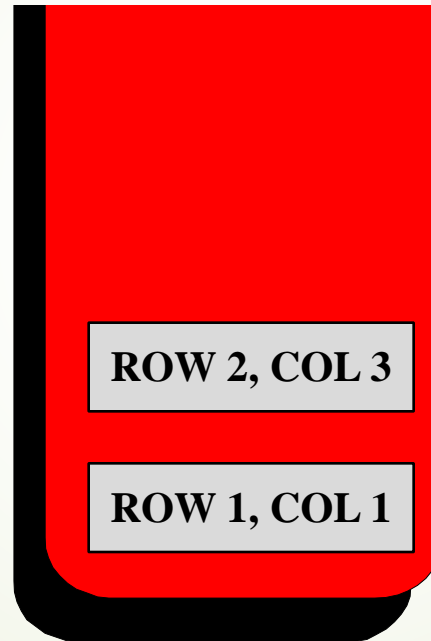
2

filled

# How the program works

When we run out of room in a row:

- pop the stack,
- reduce filled by 1
- and continue working on the previous row.



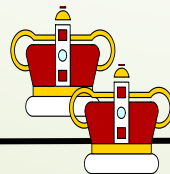
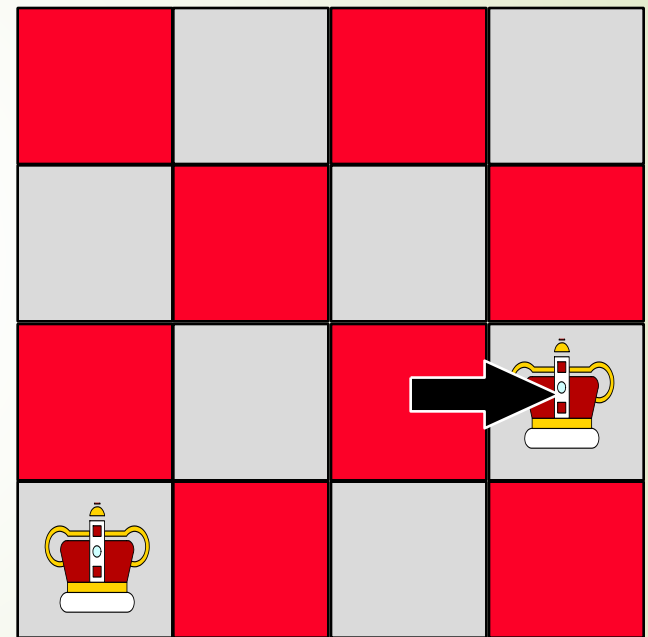
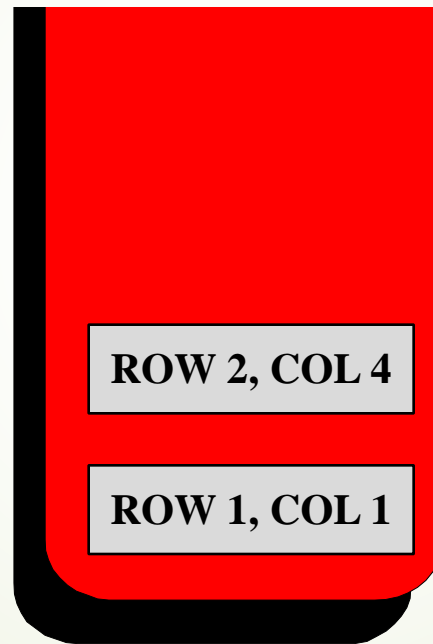
1

filled



# How the program works

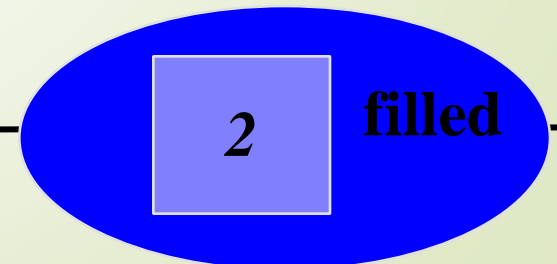
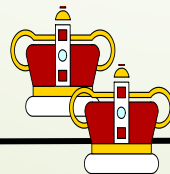
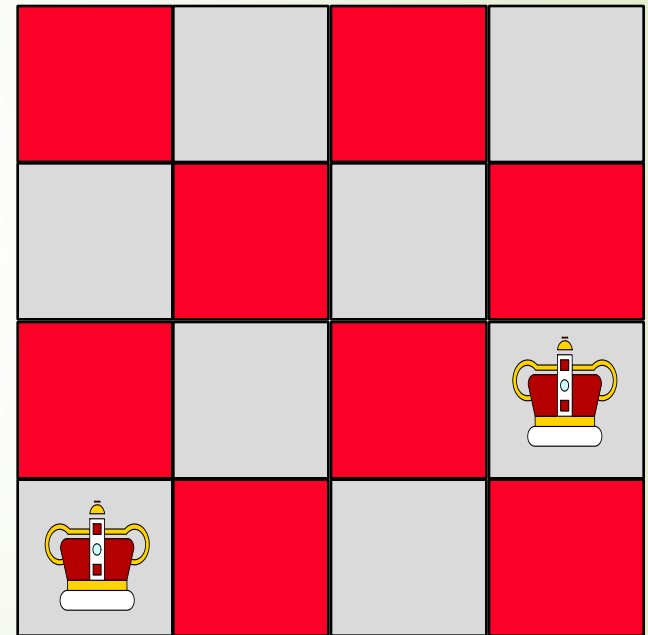
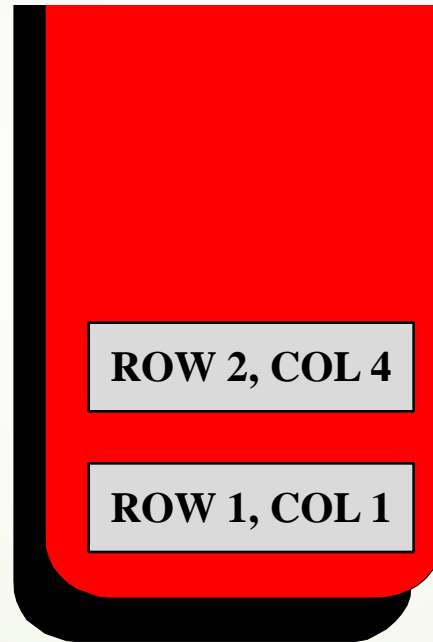
Now we continue working on row 2, shifting the queen to the right.



filled

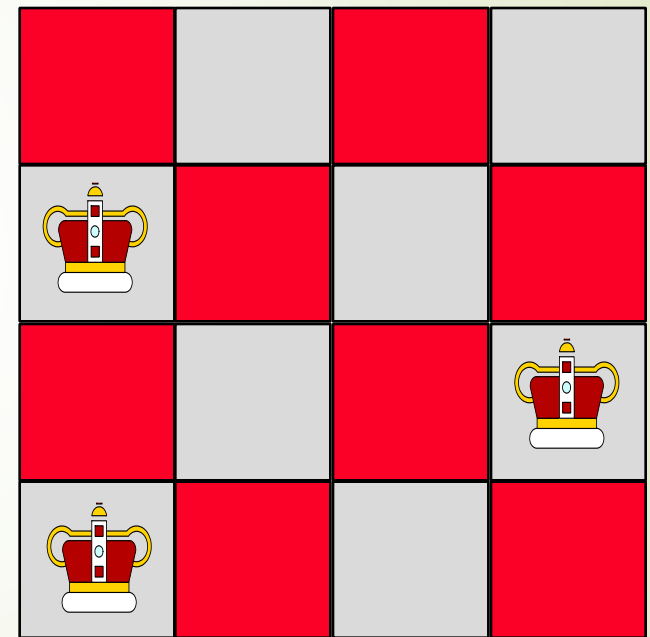
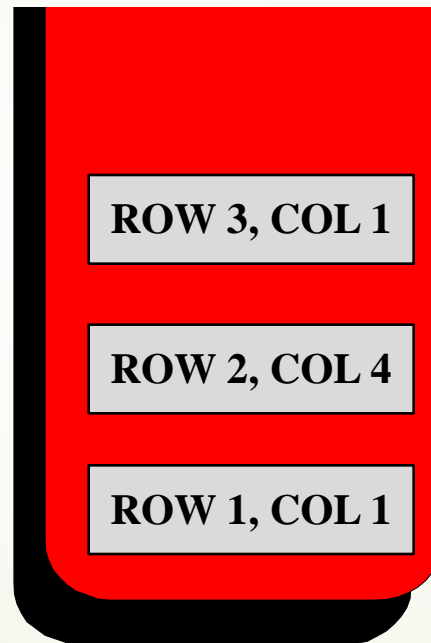
# How the program works

This position has no conflicts, so we can increase filled by 1, and move to row 3.



# How the program works

In row 3, we start again at the first column.

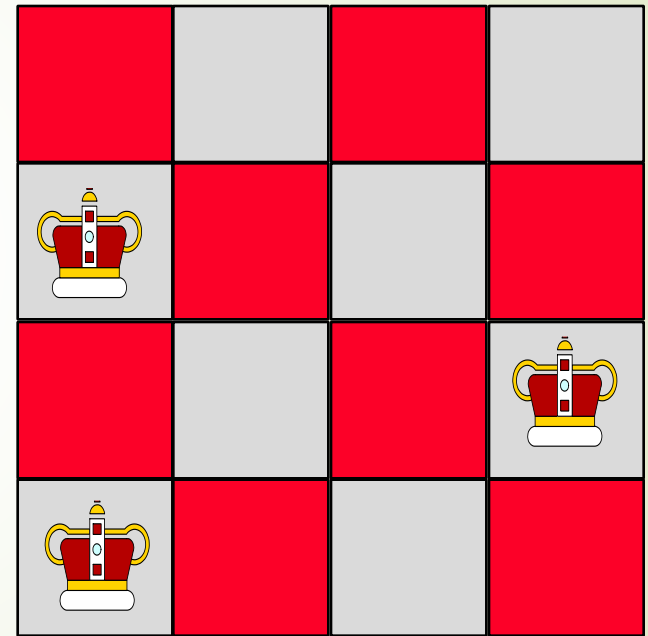
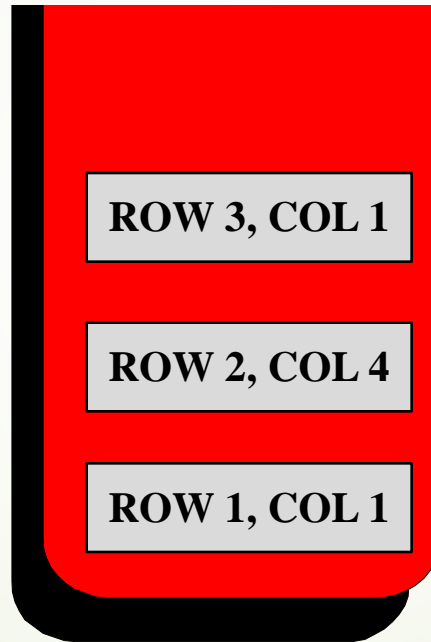


2

filled

# How the program works

In row 3, we start again at the first column.

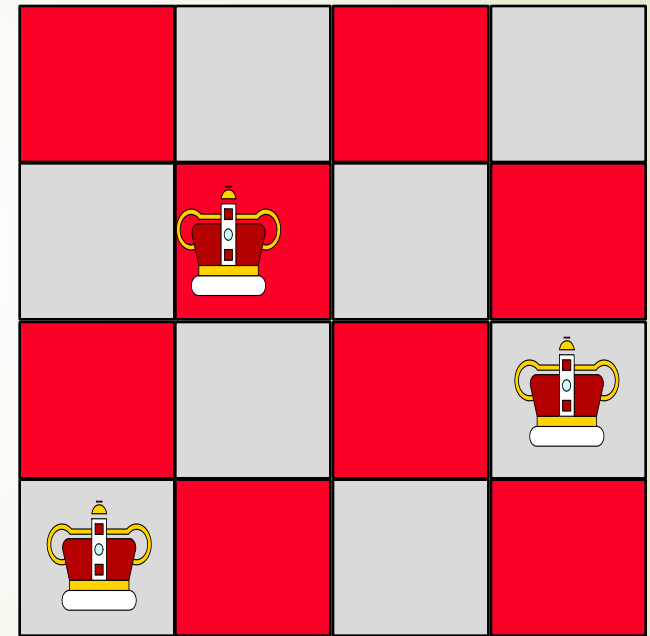
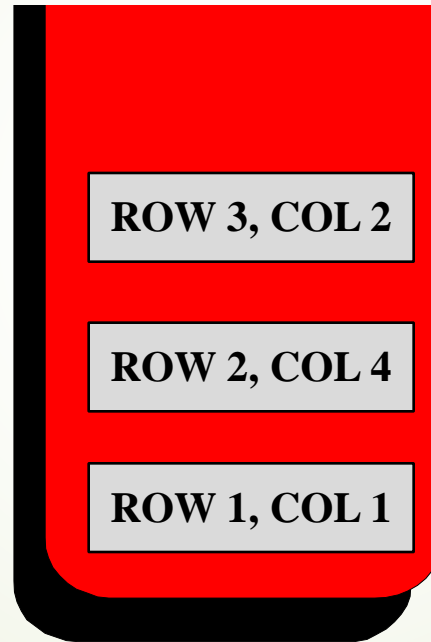


2

filled

# How the program works

In row 3, we start again at the 2<sup>nd</sup> column.

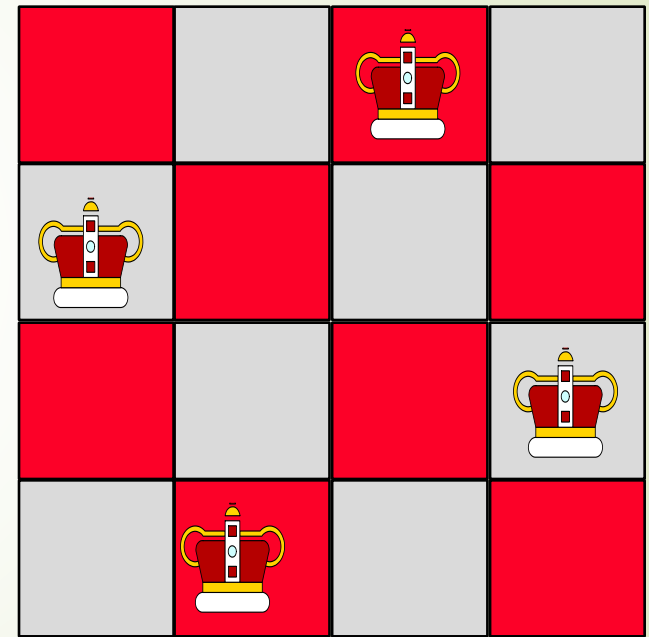
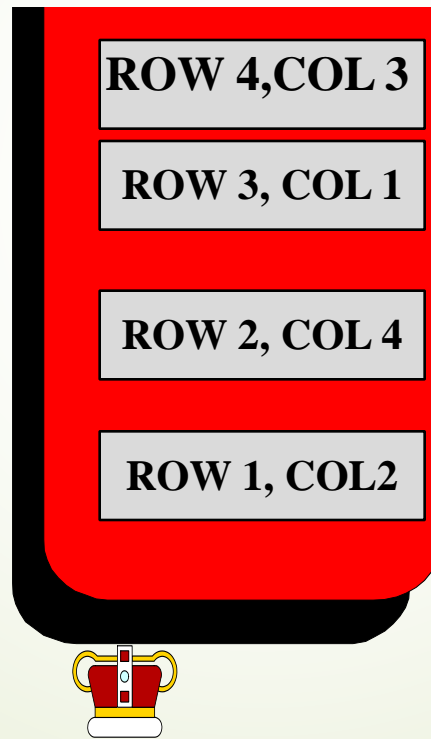


2

filled

# How the program works

In row 4, we start again at the first column.



2

filled





# How the program works

SOLVED!!!

ROW 3, COL 2

ROW 2, COL 4

ROW 1, COL 1



2

filled

# Pseudocode for N-Queens

- Initialize a stack where we can keep track of our decisions.
  - Place the first queen, pushing its position onto the stack and setting filled to 0.
  - repeat these steps
    - ▶ if there are no conflicts with the queens...
    - ▶ else if there is a conflict and there is room to shift the current queen rightward...
    - ▶ else if there is a conflict and there is no room to shift the current queen rightward...
-



# Pseudocode for N-Queens

- repeat these steps
  - if there are no conflicts with the queens...

Increase filled by 1. If filled is now N, then the algorithm is done. Otherwise, move to the next row and place a queen in the first column.

---

# Pseudocode for N-Queens

□ repeat these steps

- ▶ if there are no conflicts with the queens...
- ▶ else if there is a conflict and there is room to shift the current queen rightward...

Move the current queen rightward,  
adjusting the record on top of the stack  
to indicate the new position.

---

# Pseudocode for N-Queens

## □ repeat these steps

- ▶ if there are no conflicts with the queens...
- ▶ else if there is a conflict and there is room to shift the current queen rightward...
- ▶ else if there is a conflict and there is no room to shift the current queen rightward...

### Backtrack!

Keep popping the stack, and reducing filled by 1, until you reach a row where the queen can be shifted rightward. Shift this queen right.

# Pseudocode for N-Queens

## □ repeat these steps

- ▶ if there are no conflicts with the queens...
- ▶ else if there is a conflict and there is room to shift the current queen rightward...
- ▶ else if there is a conflict and there is no room to shift the current queen rightward...

### Backtrack!

Keep popping the stack, and reducing filled by 1, **until you reach a row where the queen can be shifted rightward.** Shift this queen right.



# Summary

- ❑ Stacks have many applications.
  - ❑ The application which we have shown is called **backtracking**.
  - ❑ The key to backtracking: Each choice is recorded in a stack.
  - ❑ When you run out of choices for the current decision, you pop the stack, and continue trying different choices for the previous decision.
-



## WEBSITES

1. [www.mit.edu](http://www.mit.edu)
  2. [www.soe.stanford.edu](http://www.soe.stanford.edu)
  3. [www.grad.gatech.edu](http://www.grad.gatech.edu)
  4. [www.gsas.harvard.edu](http://www.gsas.harvard.edu)
  5. [www.eng.ufl.edu](http://www.eng.ufl.edu)
  6. [www.iitk.ac.in](http://www.iitk.ac.in)
  7. [www.iitd.ernet.in](http://www.iitd.ernet.in)
  8. [www.ieee.org](http://www.ieee.org)
  9. [www.ntpel.com](http://www.ntpel.com)
  10. [WWW.JNTUWORLD.COM](http://WWW.JNTUWORLD.COM)
  11. [www.firstrankers.com](http://www.firstrankers.com)
  12. [www.studentgalaxi.blogspot.com](http://www.studentgalaxi.blogspot.com)
-



## **SUGGESTED BOOKS**

### **TEXT BOOKS**

1. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahni and Rajasekharam, Galgotia publications pvt. Ltd.
  2. Algorithm Design: Foundations, Analysis and Internet examples,  
M.T. Goodrich and R. Tomassia, John Wiley and Sons.
-



## REFERENCES

1. Introduction to Algorithms, second edition, T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, PHI Pvt. Ltd./ Pearson Education
  2. Introduction to Design and Analysis of Algorithms A strategic approach,  
R.C.T. Lee, S.S. Tseng, R.C. Chang and T. Tsai, Mc Graw Hill.
  3. Data structures and Algorithm Analysis in C++, Allen Weiss, Second edition, Pearson education.
-





Thank You

---