More Recursion

Tiziana Ligorio

tligorio@hunter.cuny.edu

Today's Plan

Recursion Review

8 Qeens Problem

Permutations

Combinations

Announcements

Midterm Exam postponed to **Friday March 22**

It will cover everything up to and including Recursion

Review requires your active participation

Types of Recursion

Reverse String:

- single recursive call
- Base case: stop => no return value

Dictionary:

- split problem into halves but solve only 1
- Base case: stop => no return value

Fractal Tree:

- split problem into halves and solve both
- Base case: stop => no return value

Factorial:

- single recursive call
- Base case: return a value for computation in each recursive call

Why/When use recursion

Usually less efficient than iterative counterparts (we will see example later in the course)

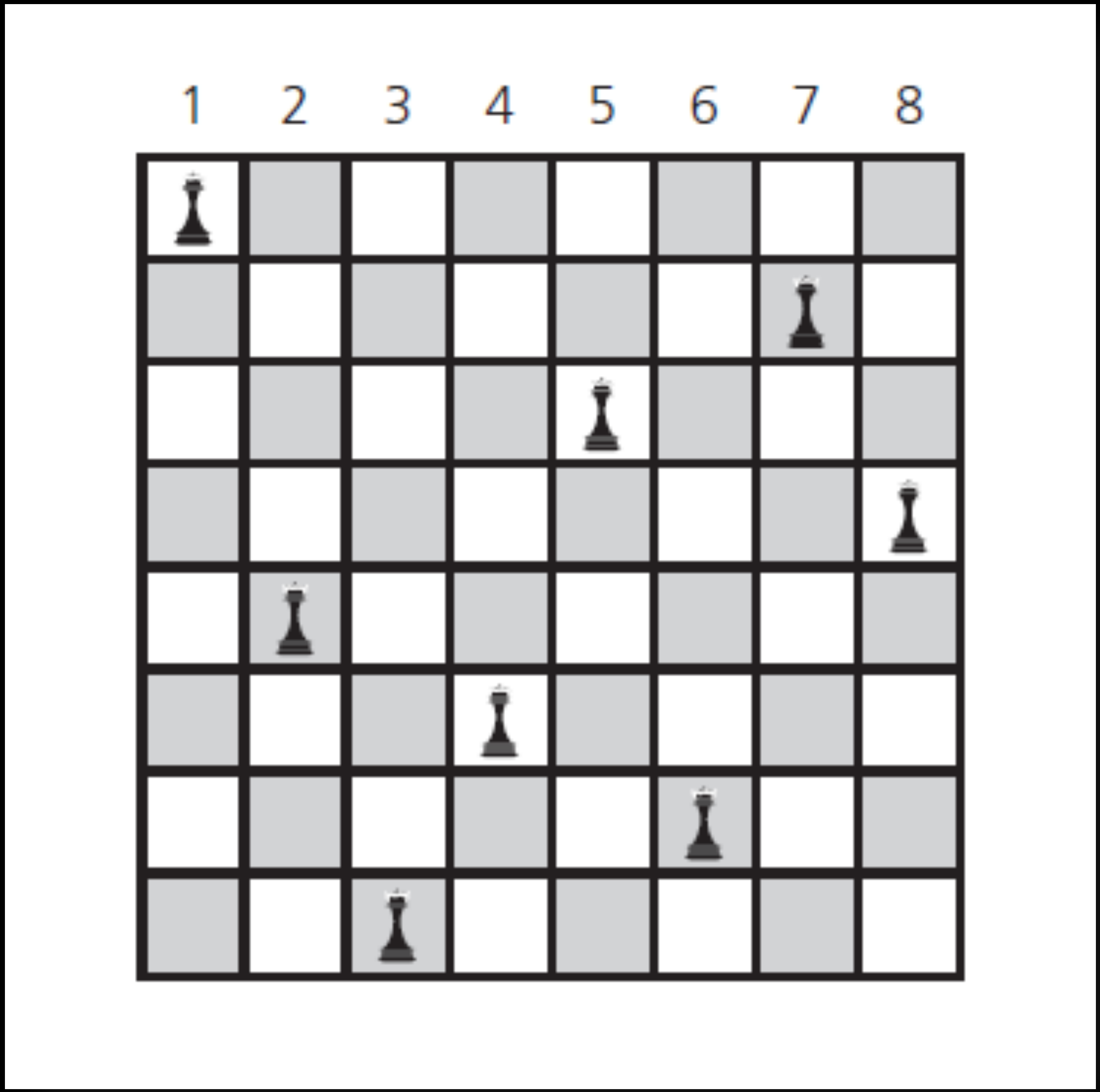
Inherent overhead associated with function calls

Repeated recursive calls with same parameters

Compilers can optimize tail-recursive (recursive call is the last statement in the function) functions to be iterative

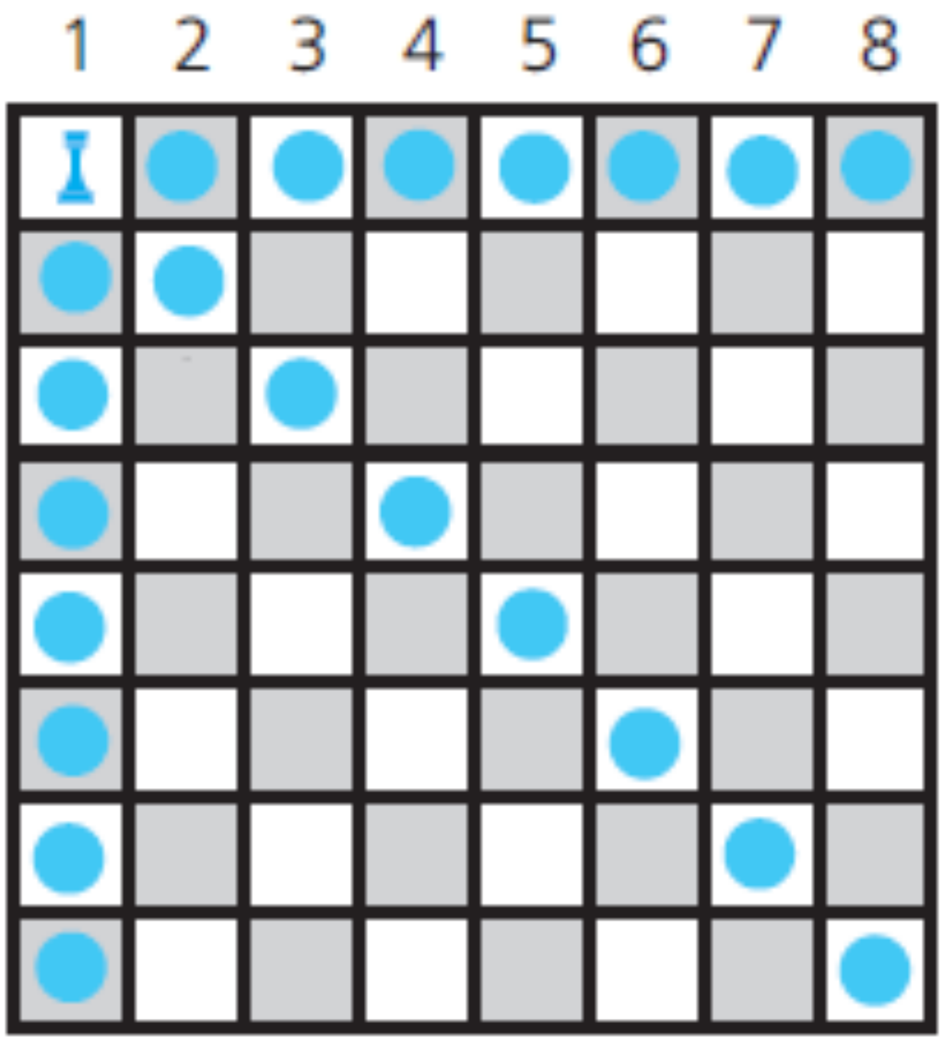
Sometimes logic of iterative solution can be very complex in comparison to recursive solution

The Eight Queens Problem



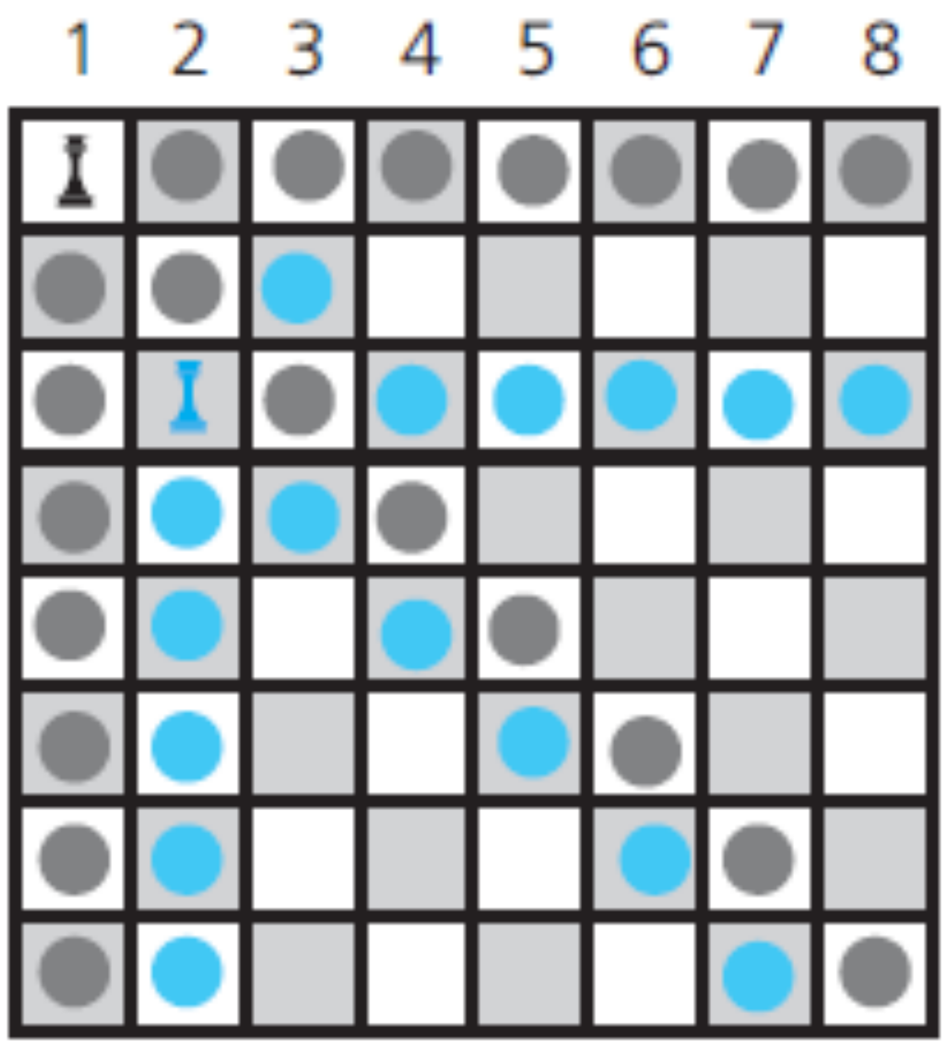
Place 8 Queens on the board s.t. no queen is on the same row, column or diagonal

The Eight Queens Problem



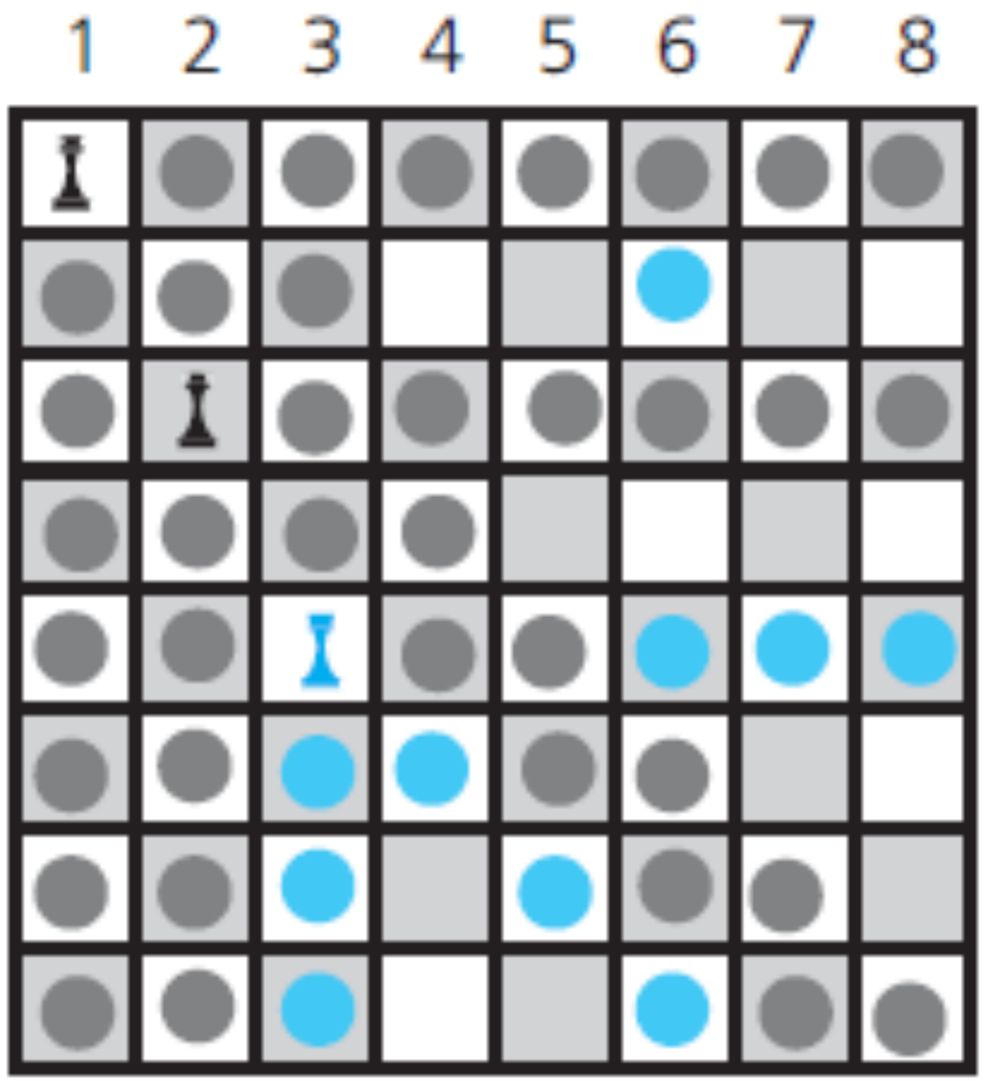
(a) The first queen in column 1

The Eight Queens Problem



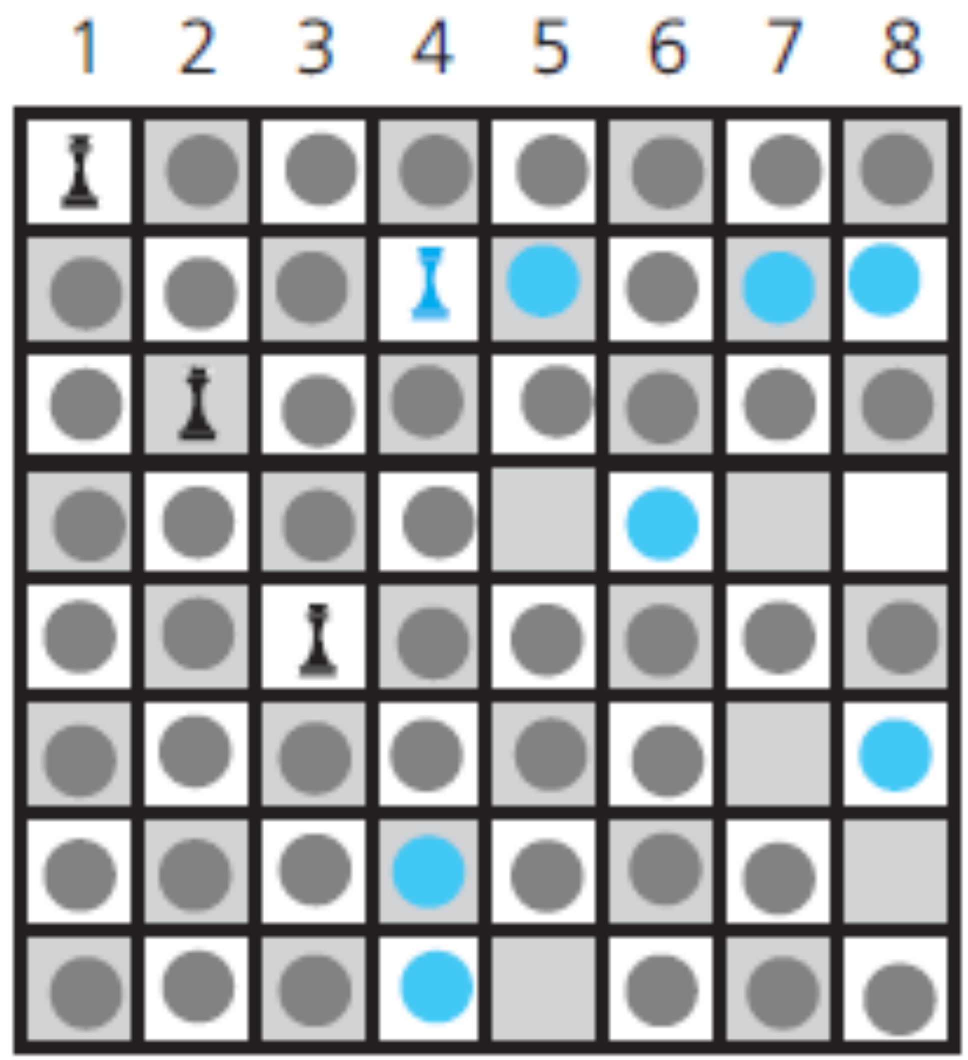
(b) The second queen in column 2

The Eight Queens Problem



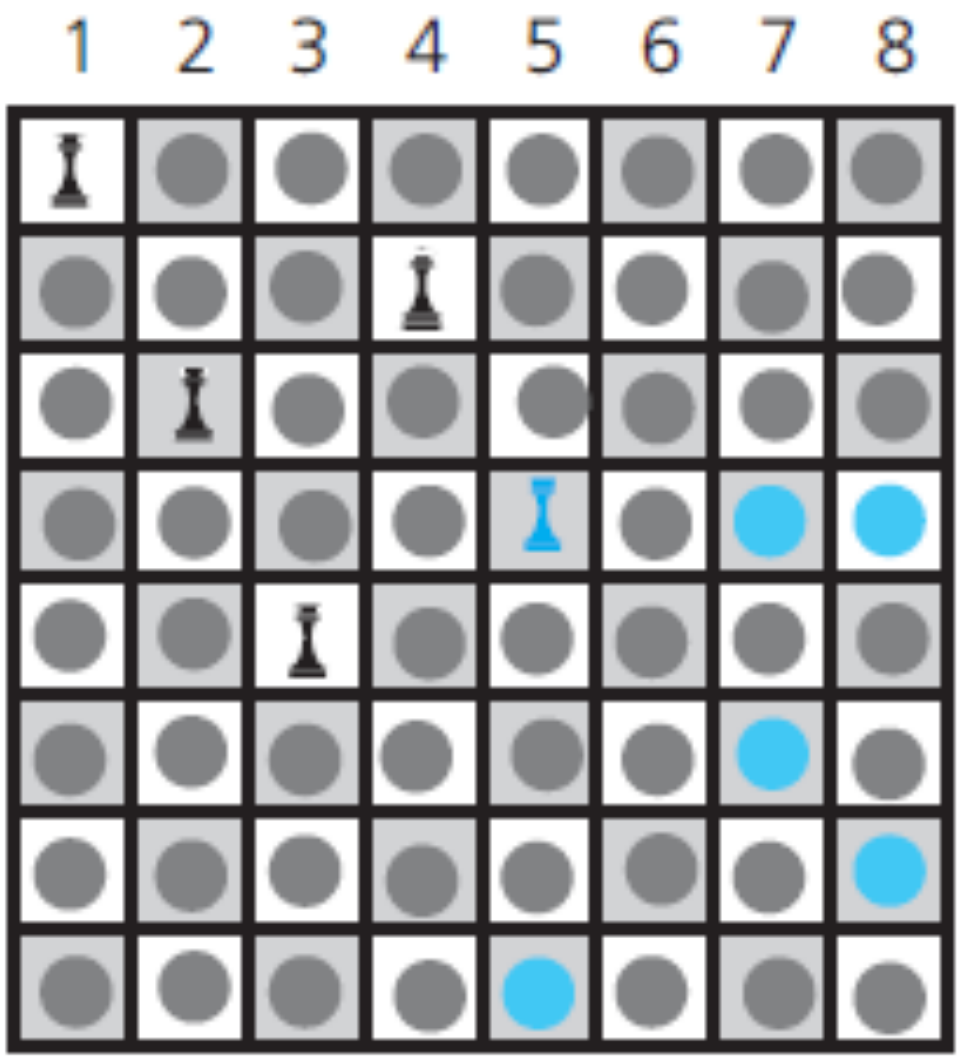
(c) The third queen in column 3

The Eight Queens Problem



(d) The fourth queen in column 4

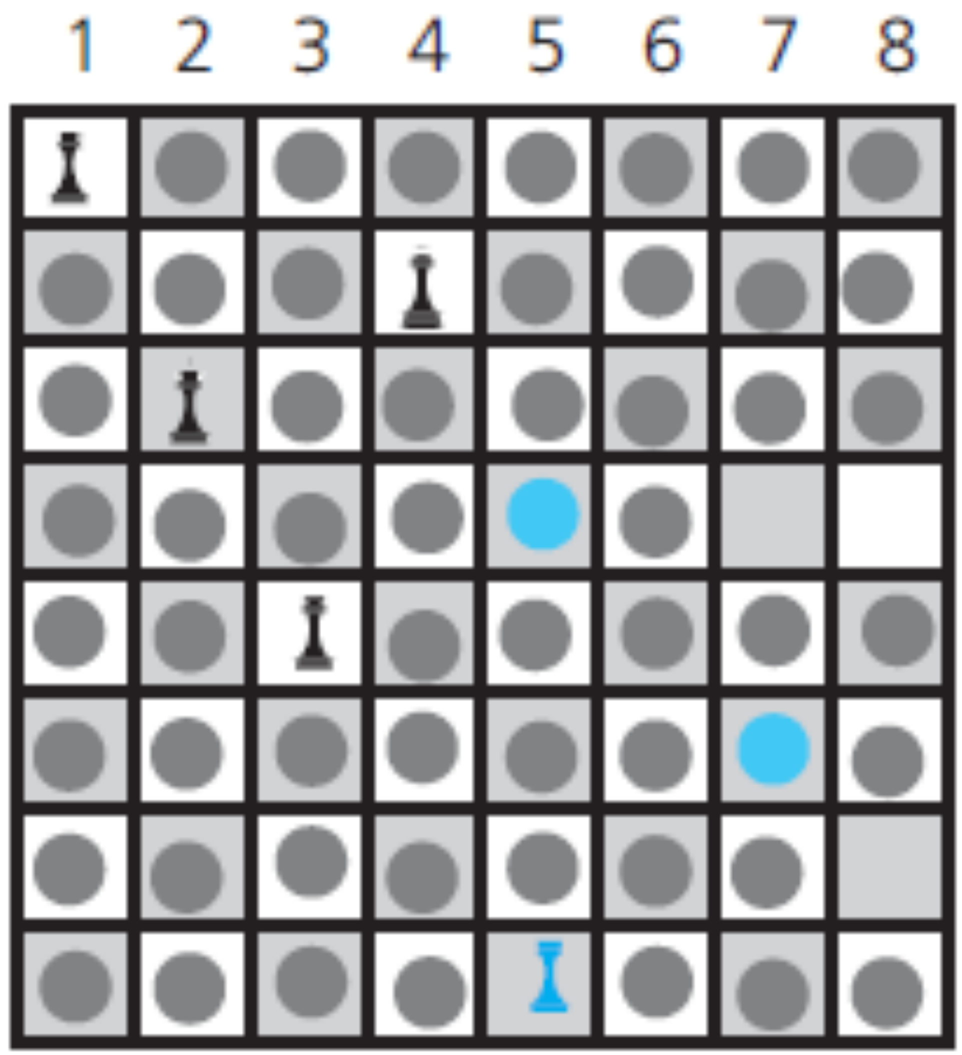
The Eight Queens Problem



(e) The five queens
can attack all of column 6

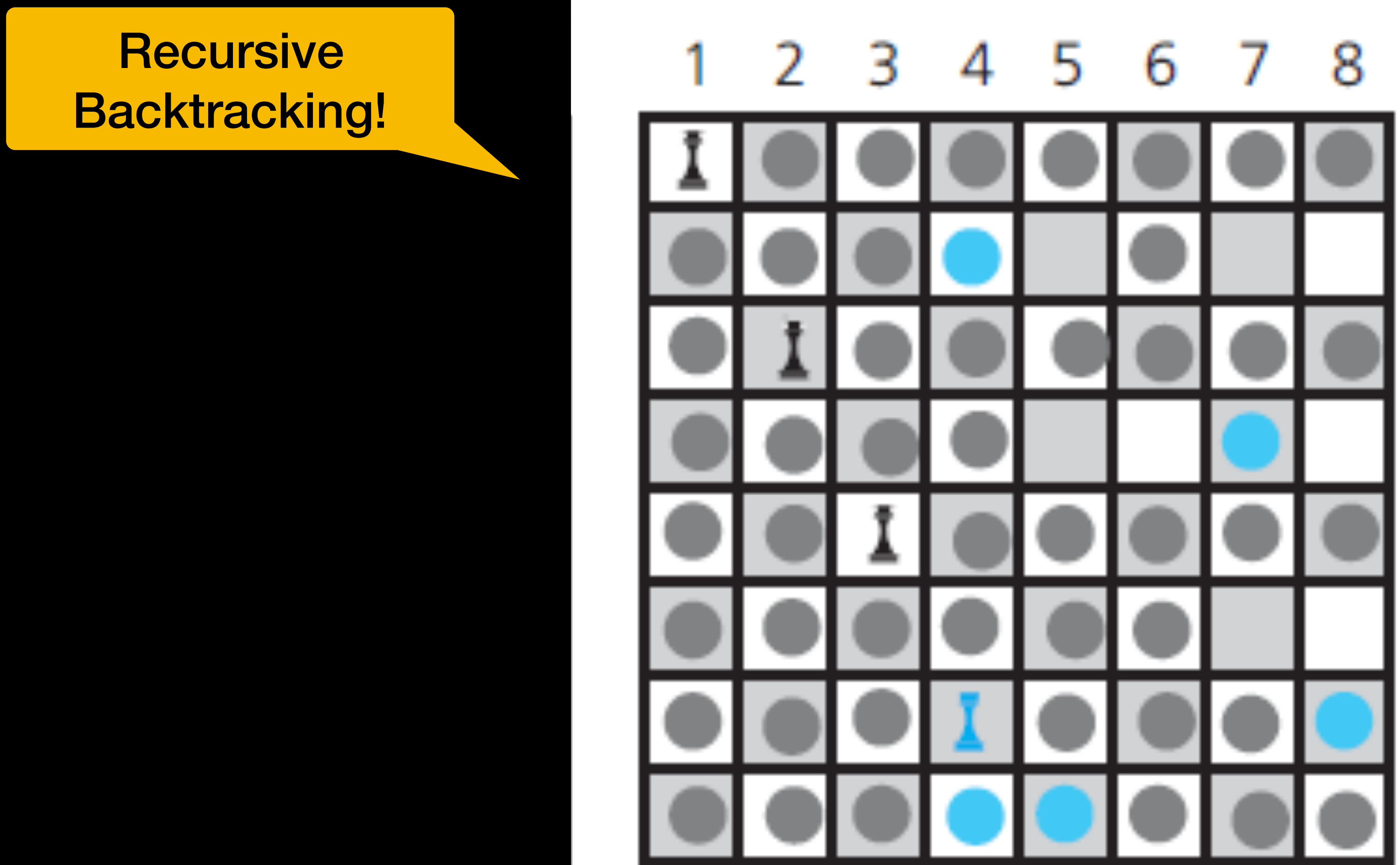
The Eight Queens Problem

Recursive Backtracking!



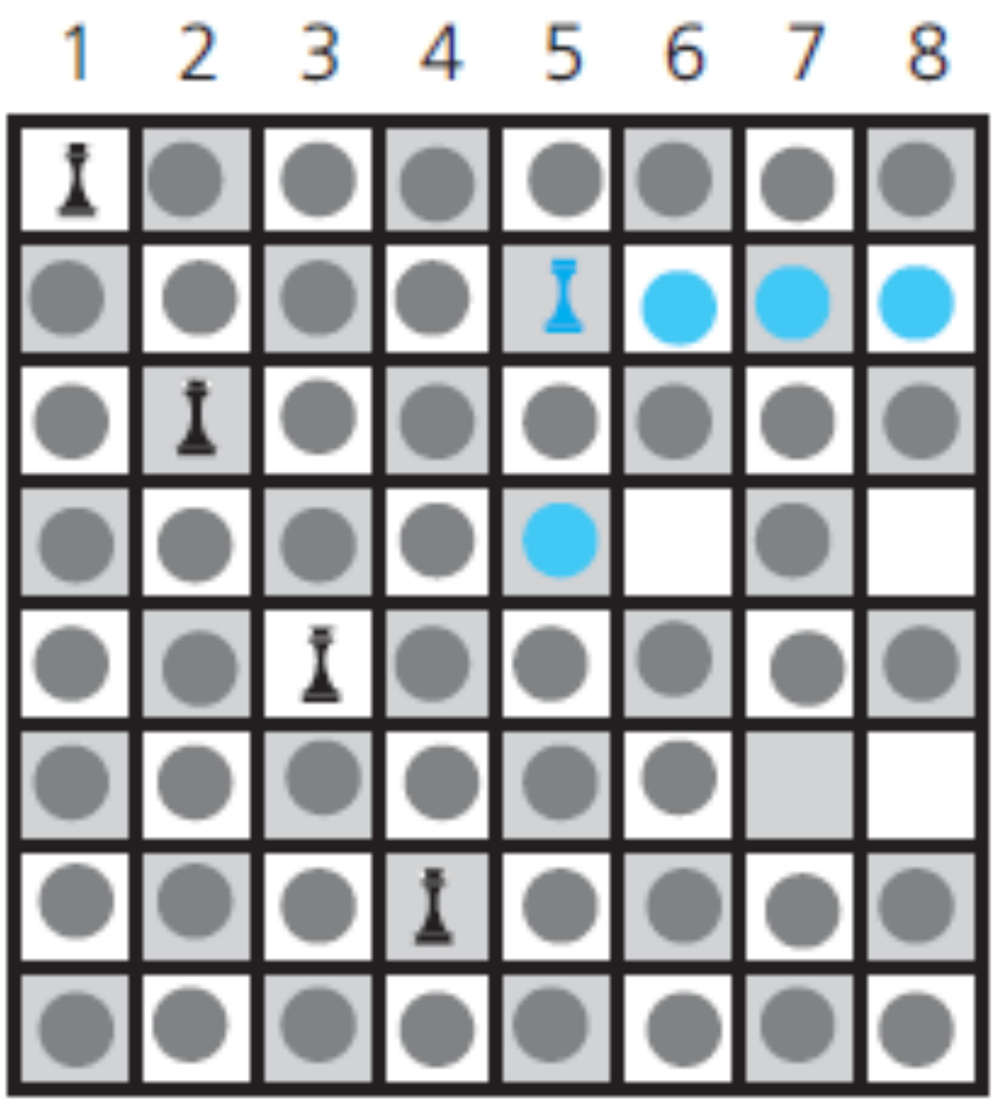
(f) Backtracking to column 5 to try another square for the queen

The Eight Queens Problem



(g) Backtracking to column 4 to try another square for the queen

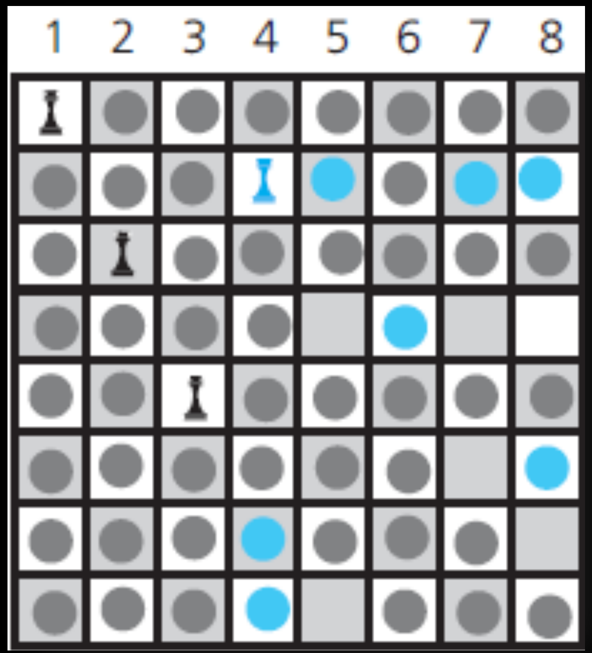
The Eight Queens Problem

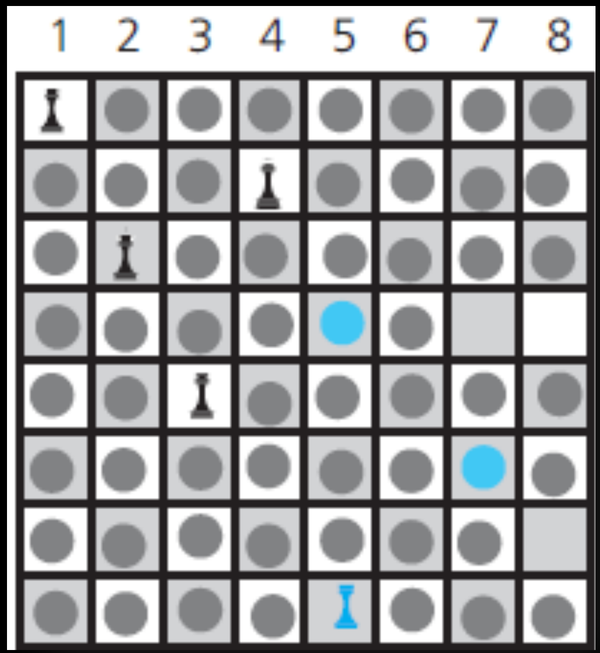


(h) Considering column 5 again

The Eight Queens Problem

```
bool placeQueens(board, column)
{
    if(column > BOARD_SIZE)
        return true; //Problem is solved!
    else
    {
        while(there are safe squares in this column)
        {
            place queen in next safe  square;
            if(placeQueen(board, column+1)) //recursively look forward
                return true; //queen safely placed
        }
        return false; //recursive backtracking
    }
}
```





Think Algorithmically

*"Experienced Computer Scientists analyze and solve computational problems at a level of **abstraction** that is beyond that of any particular programming language / representation / implementation"*

Algorithm Design

- Identify the problem
- Come up with a procedure that will lead to solution
- Independent of implementation detail

Initial phase/step

Model your problem/data

- **represent** the problem to support your algorithm

Implement solution

- Language
- Data structure
- Implementation detail

Think Algorithmically

Takes practice

The more you see/do the easier it gets

There are some frameworks that can guide you
E.g.

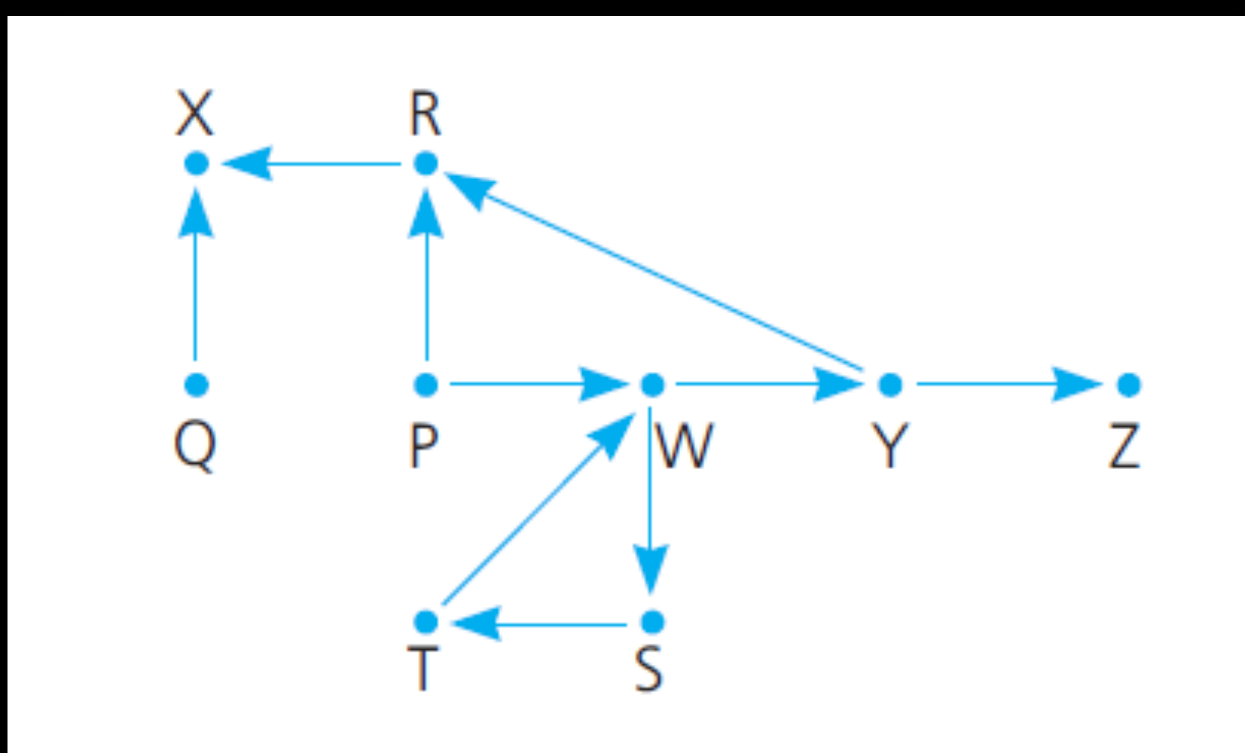
- Can I cast this as a backtracking problem?
- Can I cast this as a decision-making / decision tree problem?

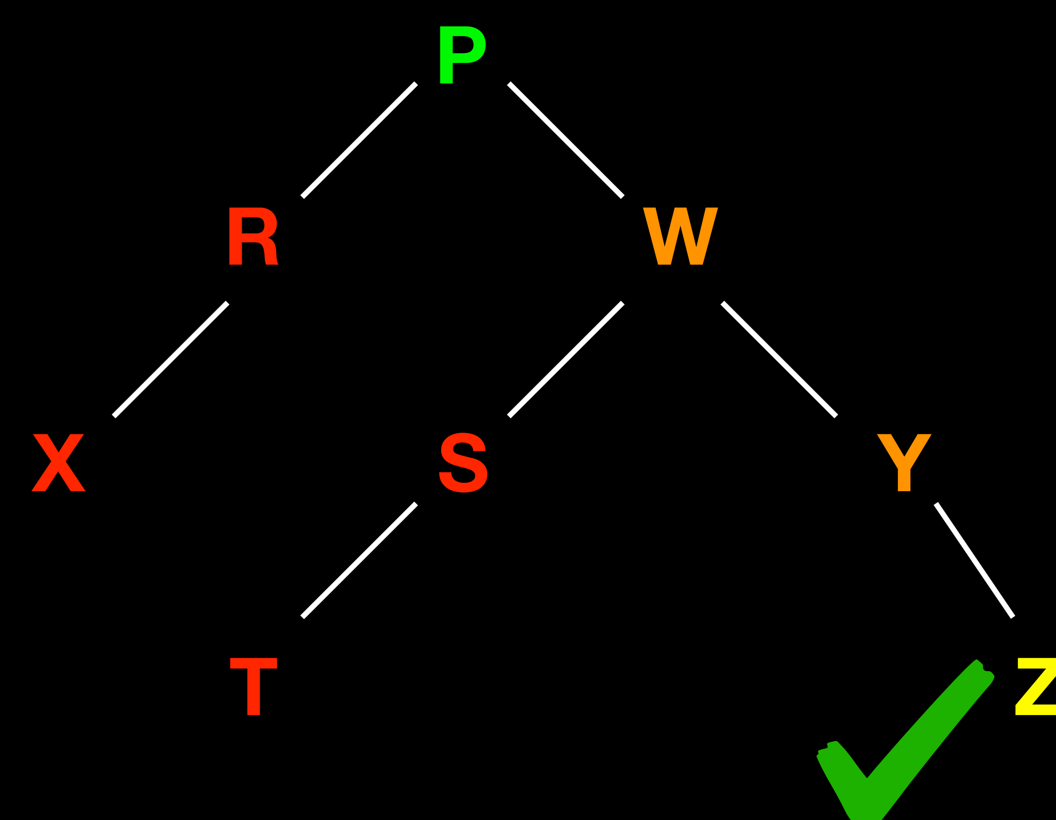
Lecture Activity

Write **PSEUDOCODE** for a **RECURSIVE** function that finds a path from origin to destination

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z





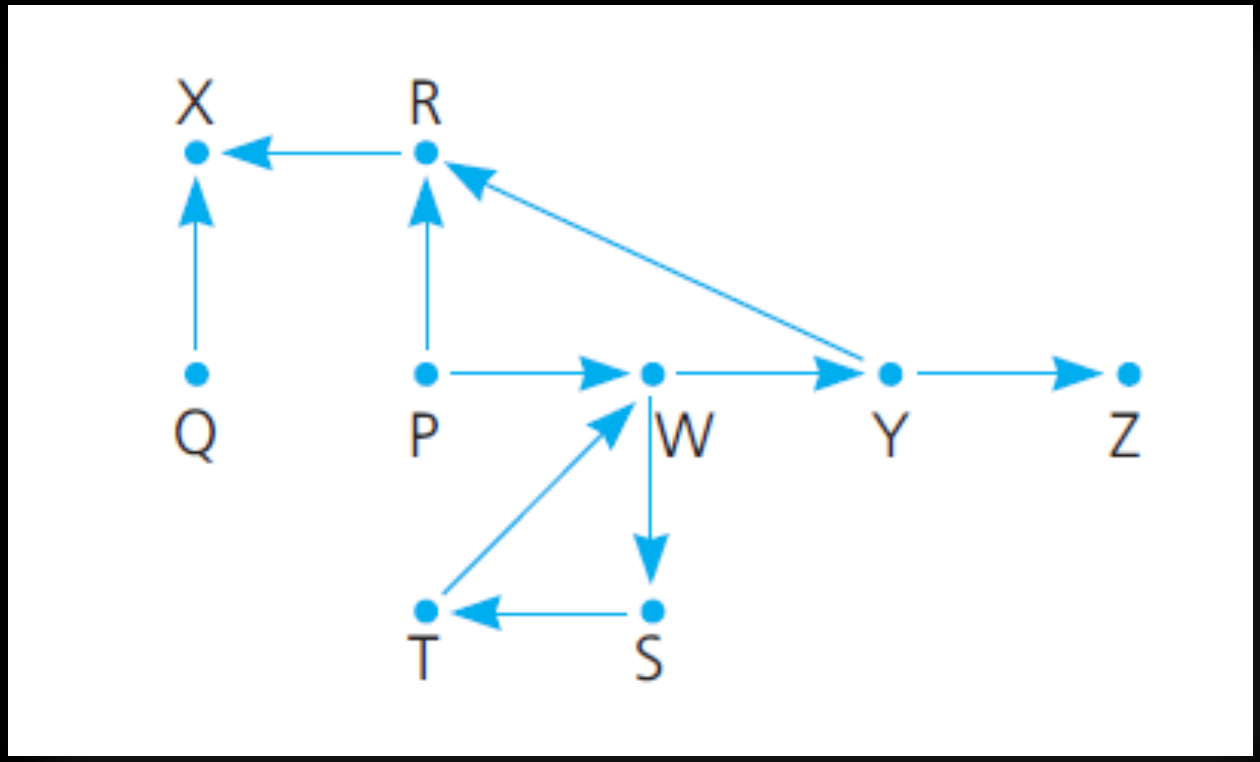
Lecture Activ

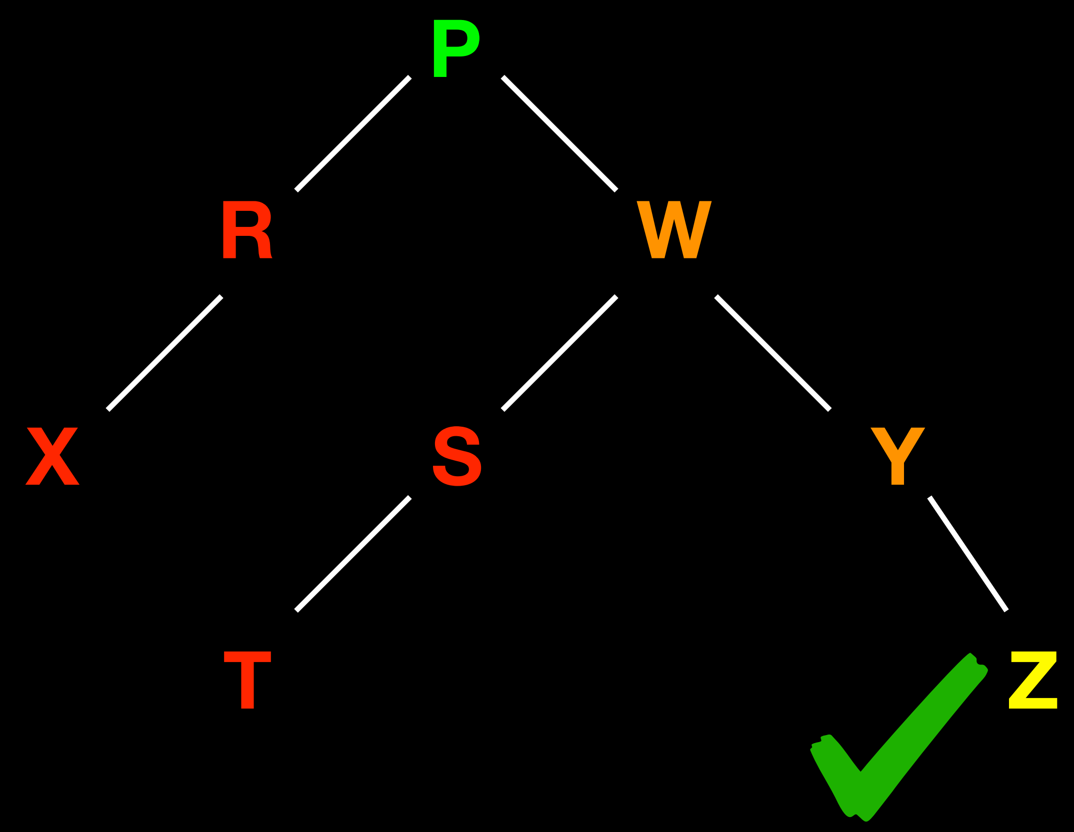
Don't get bogged down by what a map is in design phase
You know it's available and you can look up where you can go next from origin

Write **PSEUDOCODE** for a **RECURSIVE** function that finds a

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z

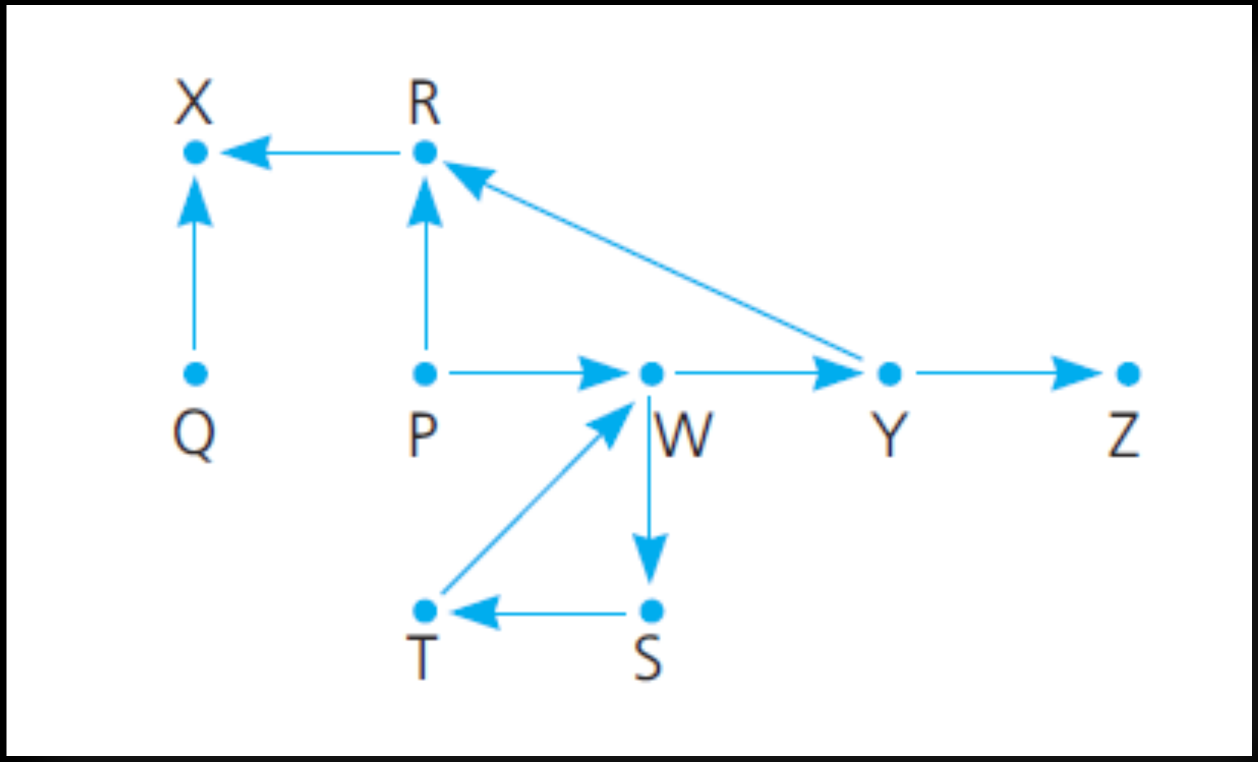


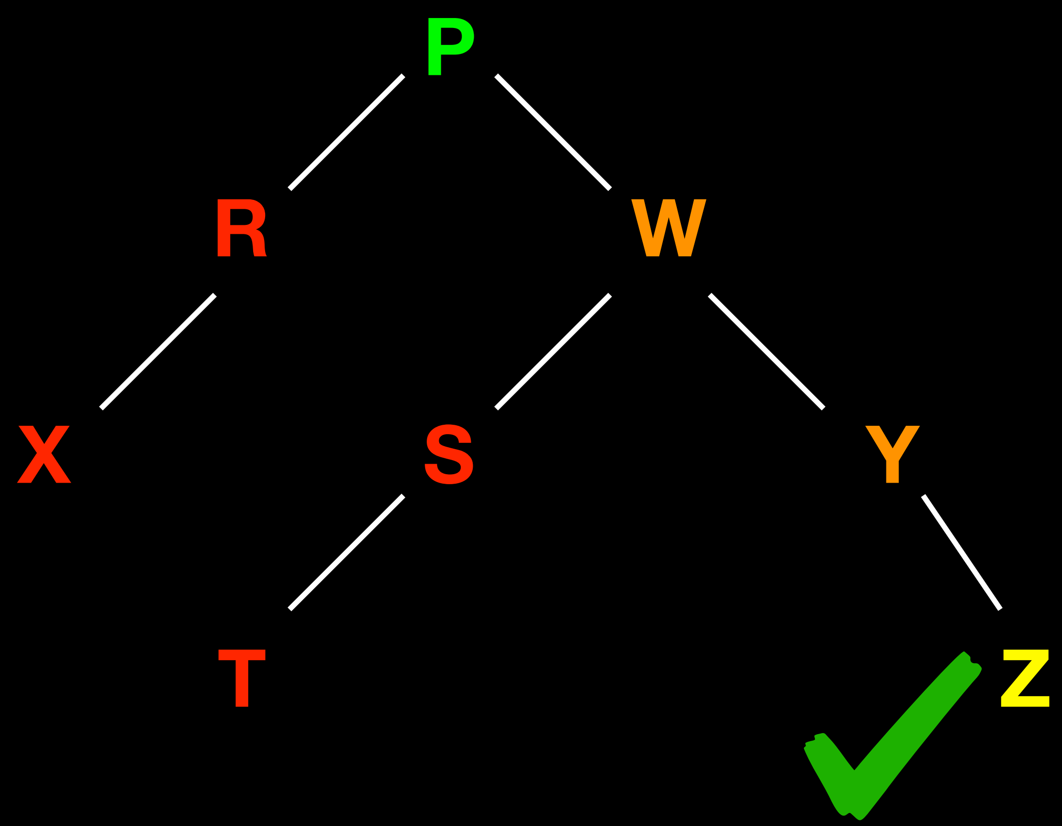


Lecture Activity

```
bool findPath(map, origin, destination)
{
     mark origin as visited in map
     if origin == destination
          return true
     else
          for each unvisited city C reachable from origin
               if findPath(map, C, destination)
                    return true
          return false //recursive backtracking
}
```

Origin = P , Destination = Z





Find Permutations

Toy example to make initial observation

Order Matters!

A	B	C	D

A	B	C	D	B	A	C	D	C	A	B	D	D	A	B	C
A	B	D	C	B	A	D	C	C	A	D	B	D	A	C	B
A	C	B	D	B	C	A	D	C	B	A	D	D	B	A	C
A	C	D	B	B	C	D	A	C	B	D	A	D	B	C	A
A	D	B	C	B	D	A	C	C	D	A	B	D	C	A	B
A	D	C	B	B	D	C	A	C	D	B	A	D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Find Permutations

Very similar to
000 100
001 101
010 110
011 111

A	B	C	D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

B	A	C	D
B	A	D	C
B	C	A	D
B	C	D	A
B	D	A	C
B	D	C	A

C	A	B	D
C	A	D	B
C	B	A	D
C	B	D	A
C	D	A	B
C	D	B	A

D	A	B	C
D	A	C	B
D	B	A	C
D	B	C	A
D	C	A	B
D	C	B	A

Lock the first letter
For each letter you lock
Permute the rest

Find Permutations

A	B	C	D

A	B	C	D	B	A	C	D	C	A	B	D	D	A	B	C
A	B	D	C	B	A	D	C	C	A	D	B	D	A	C	B
A	C	B	D	B	C	A	D	C	B	A	D	D	B	A	C
A	C	D	B	B	C	D	A	C	B	D	A	D	B	C	A
A	D	B	C	B	D	A	C	C	D	A	B	D	C	A	B
A	D	C	B	B	D	C	A	C	D	B	A	D	C	B	A

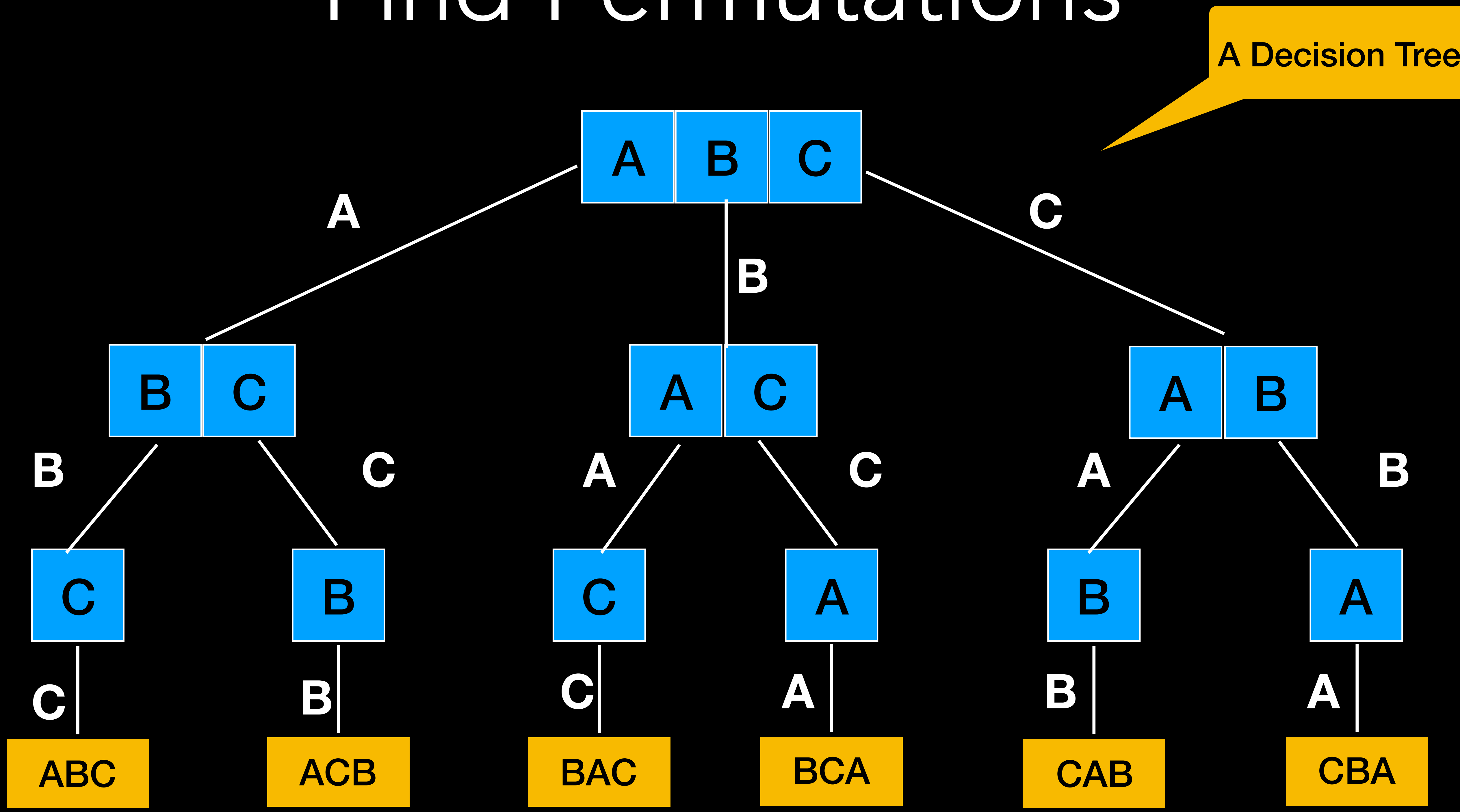
Find Permutations

Lock the first letter
For each letter you lock
Permute the rest
DECISION
RECURSION

A	B	C	D

A	B	C	D	B	A	C	D	C	A	B	D	D	A	B	C
A	B	D	C	B	A	D	C	C	A	D	B	D	A	C	B
A	C	B	D	B	C	A	D	C	B	A	D	D	B	A	C
A	C	D	B	B	C	D	A	C	B	D	A	D	B	C	A
A	D	B	C	B	D	A	C	C	D	A	B	D	C	A	B
A	D	C	B	B	D	C	A	C	D	B	A	D	C	B	A

Find Permutations



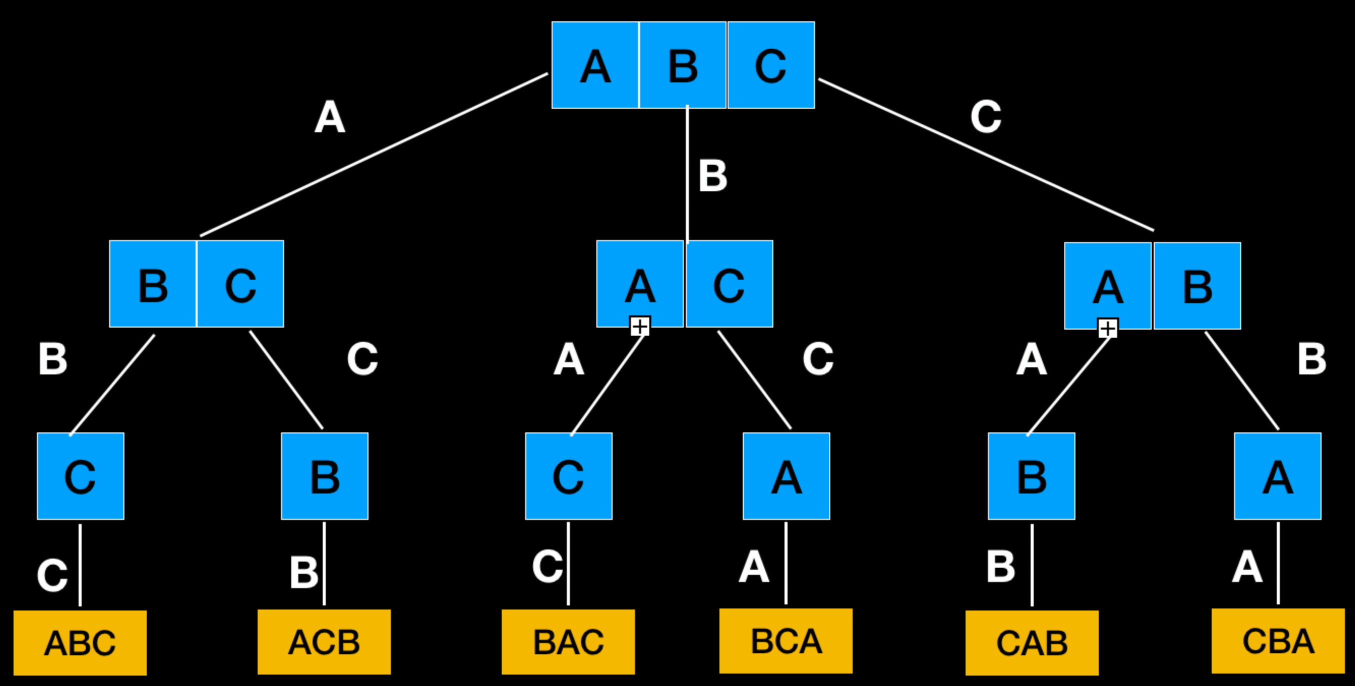
```
/**
 Prints permutations of a string
 @param str the string to be permuted
 @param l the index of the leftmost character in str substring to be permuted
 @param r the index of the rightmost character in str substring to be permuted
 */
void permuteStr(std::string str, int l, int r)
{
    if (l == r)
        std::cout << str << std::endl; //obtained one permutation to print
    else
    {
        for (int i = l; i <= r; i++)
        {
            std::swap(str[l],str[i]);//swap other characters with current first
            permuteStr(str, l+1, r);
            std::swap(str[l],str[i]); //restore first char
        }
    }
}
```

ABCD

BACD

CBAD

DBCA



Recursive Decision Tree

```
void exploreFrom(current_state, decisions_made) {
    if (all decisions have been made) { //base case
        output the result of the decisions we've made;
    } else {
        for (each decision we can make) {
            exploreFrom(result of making that decision,
                        decisions_made + this_decision);
        }
    }
}
```

Generally, if you can express a problem solution with a decision tree you can translate it into a recursive algorithm

Find Combinations

A B C D

Order does Not matter!

{}

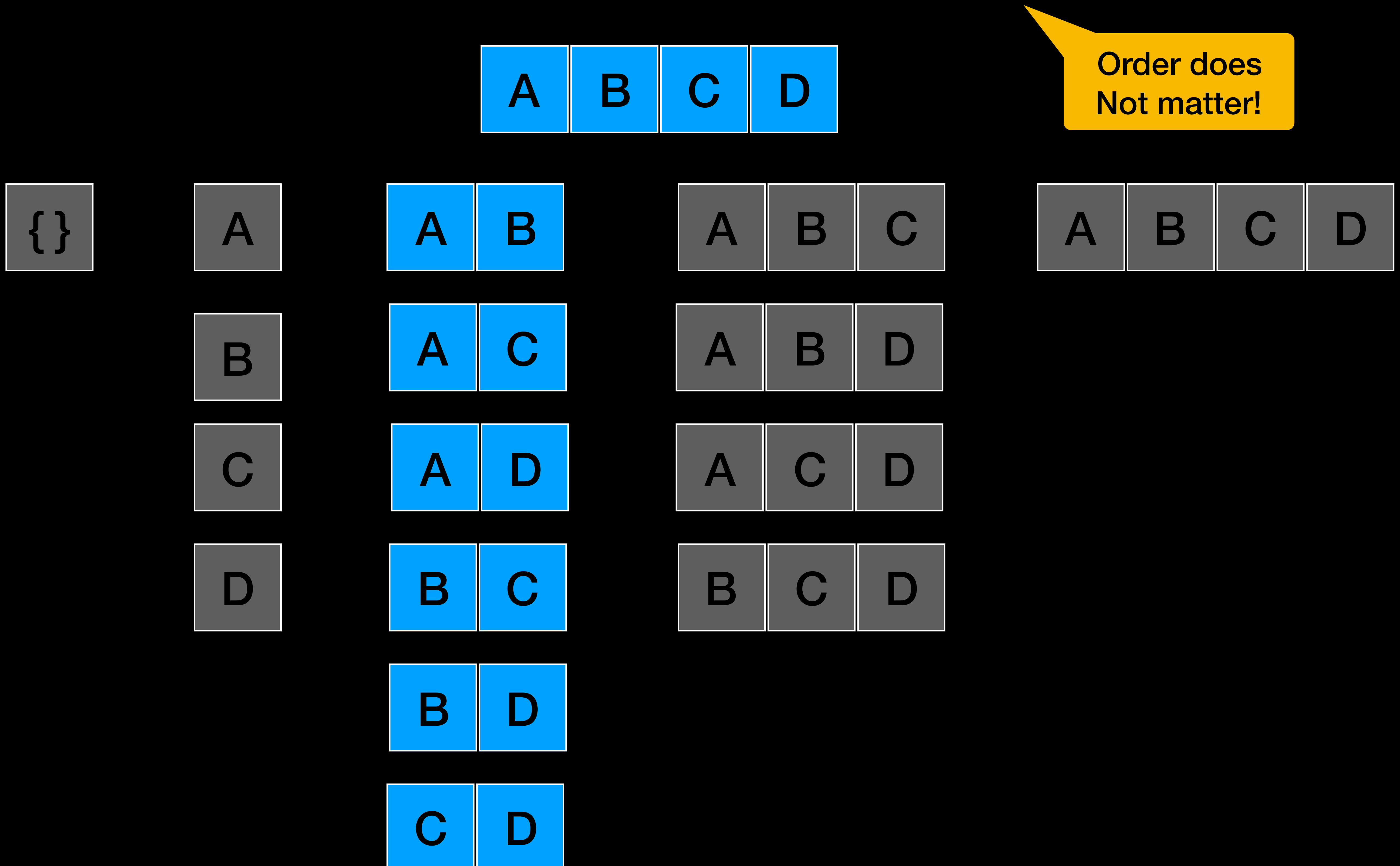
A B C D

A B, A C, A D, B C, B D, C D

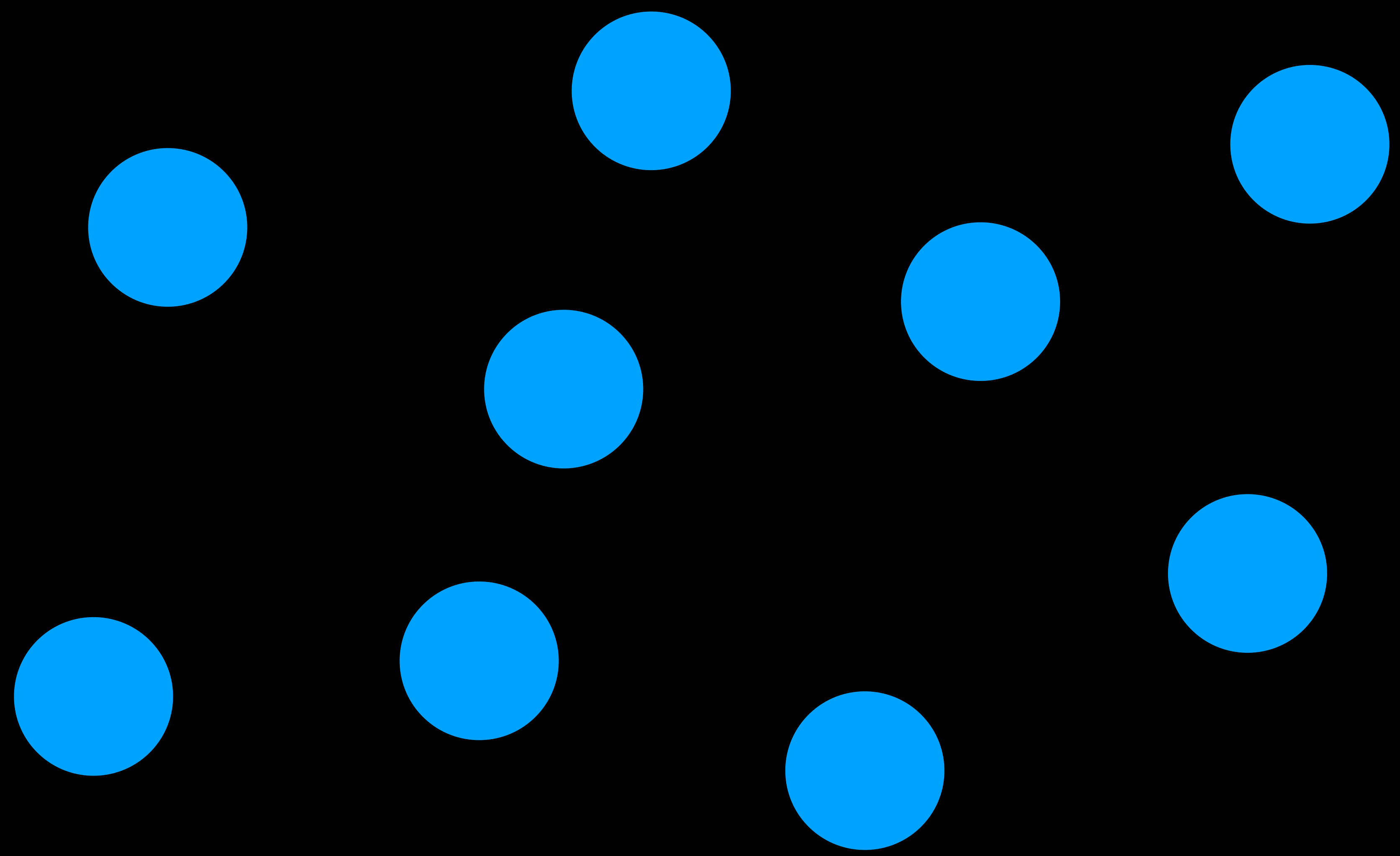
A B C, A B D, A C D, B C D

A B C D

Find Combinations of size 2



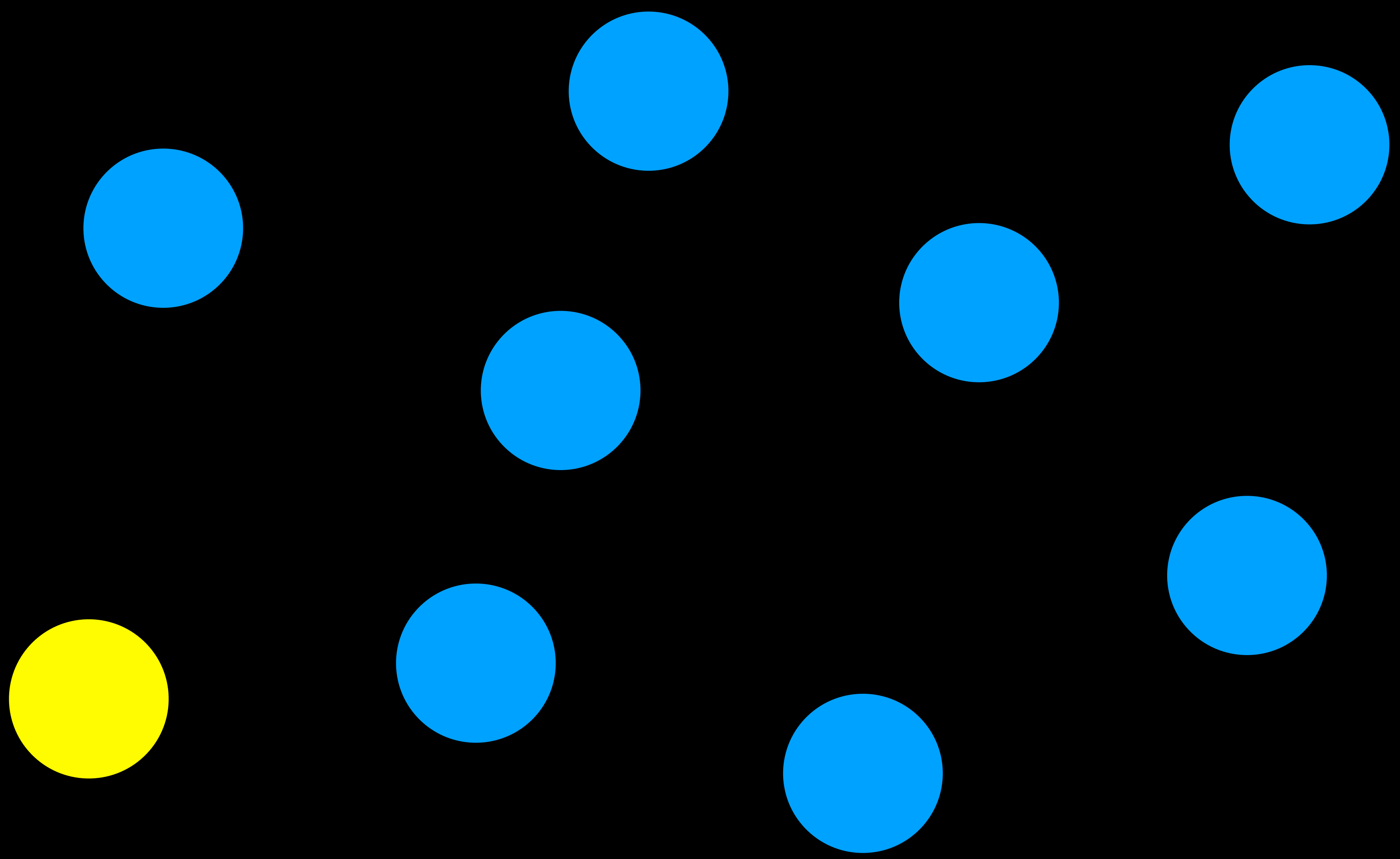
Combinations (n choose k)



One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

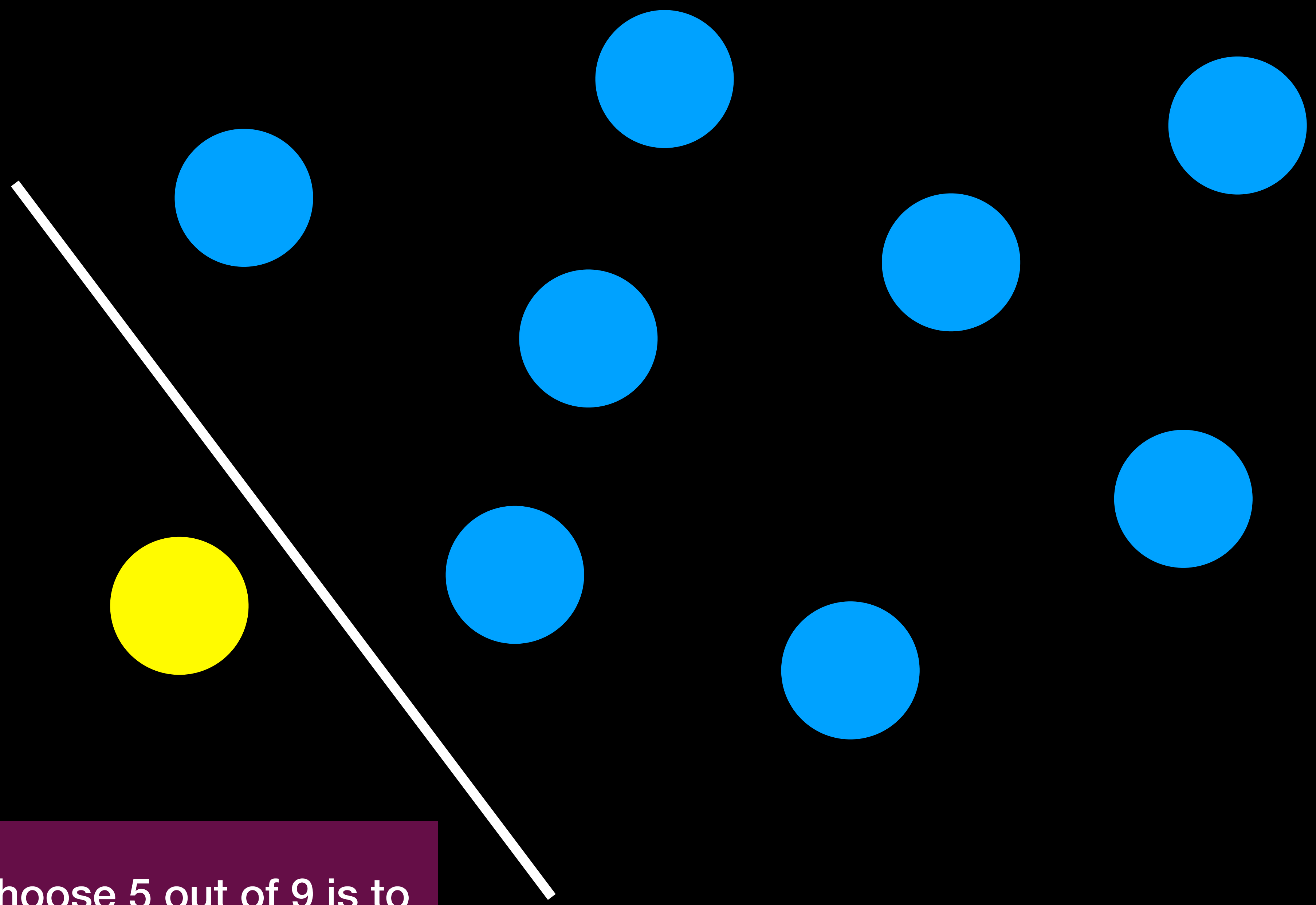
Start with toy problem
to make observation

Combinations (n choose k)



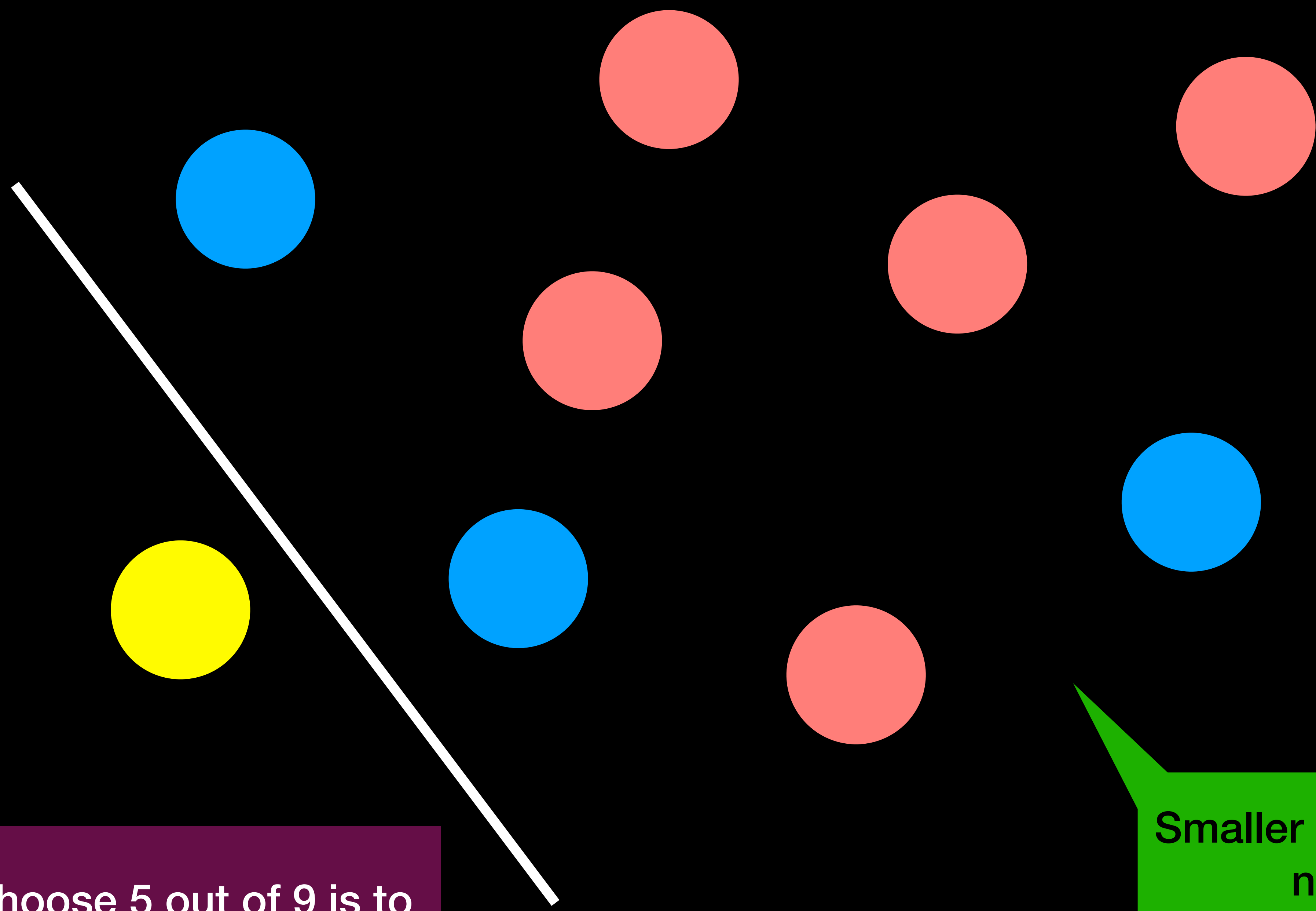
One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

Combinations (n choose k)



One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

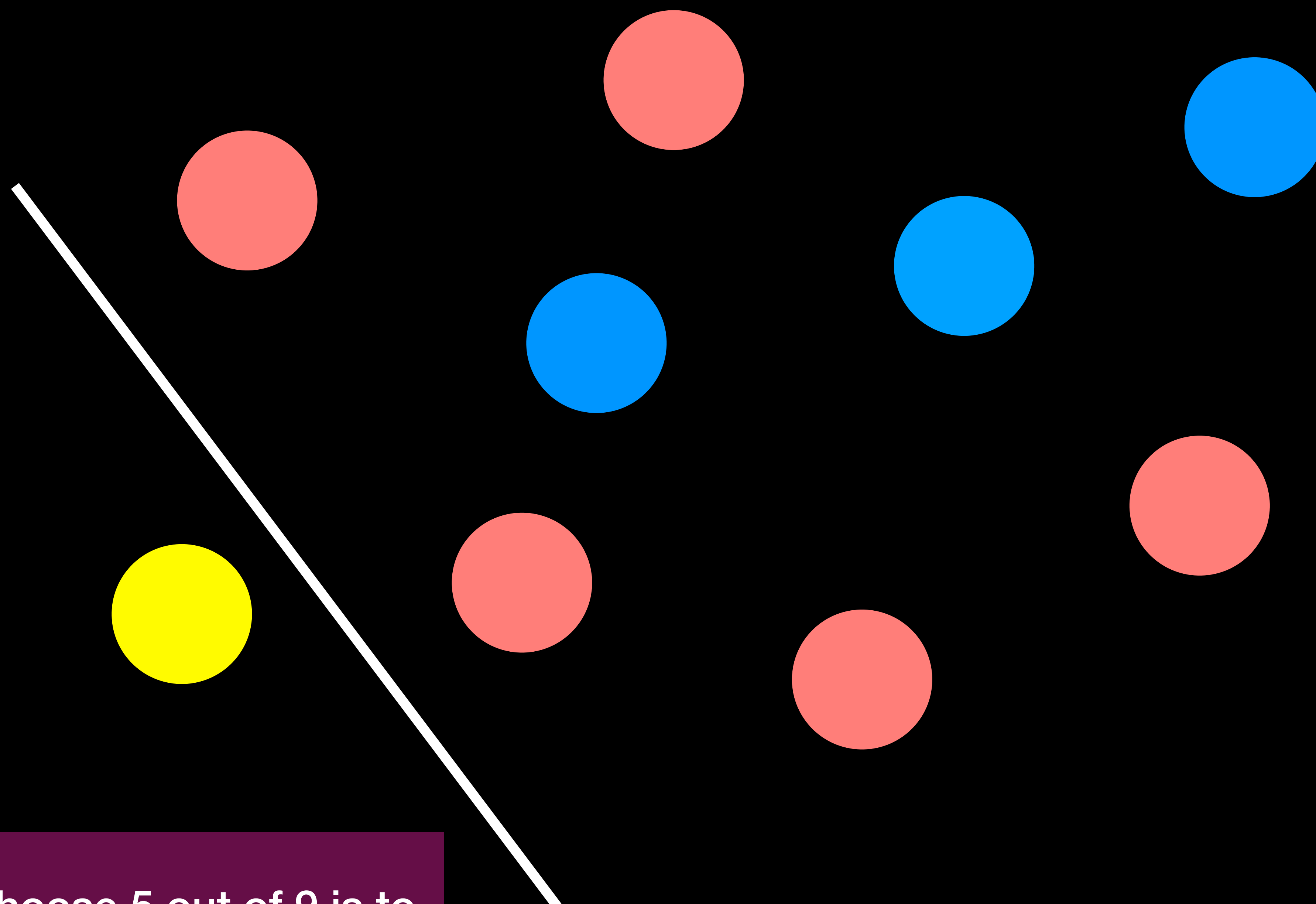
Combinations (n choose k)



Smaller problem!
n-1

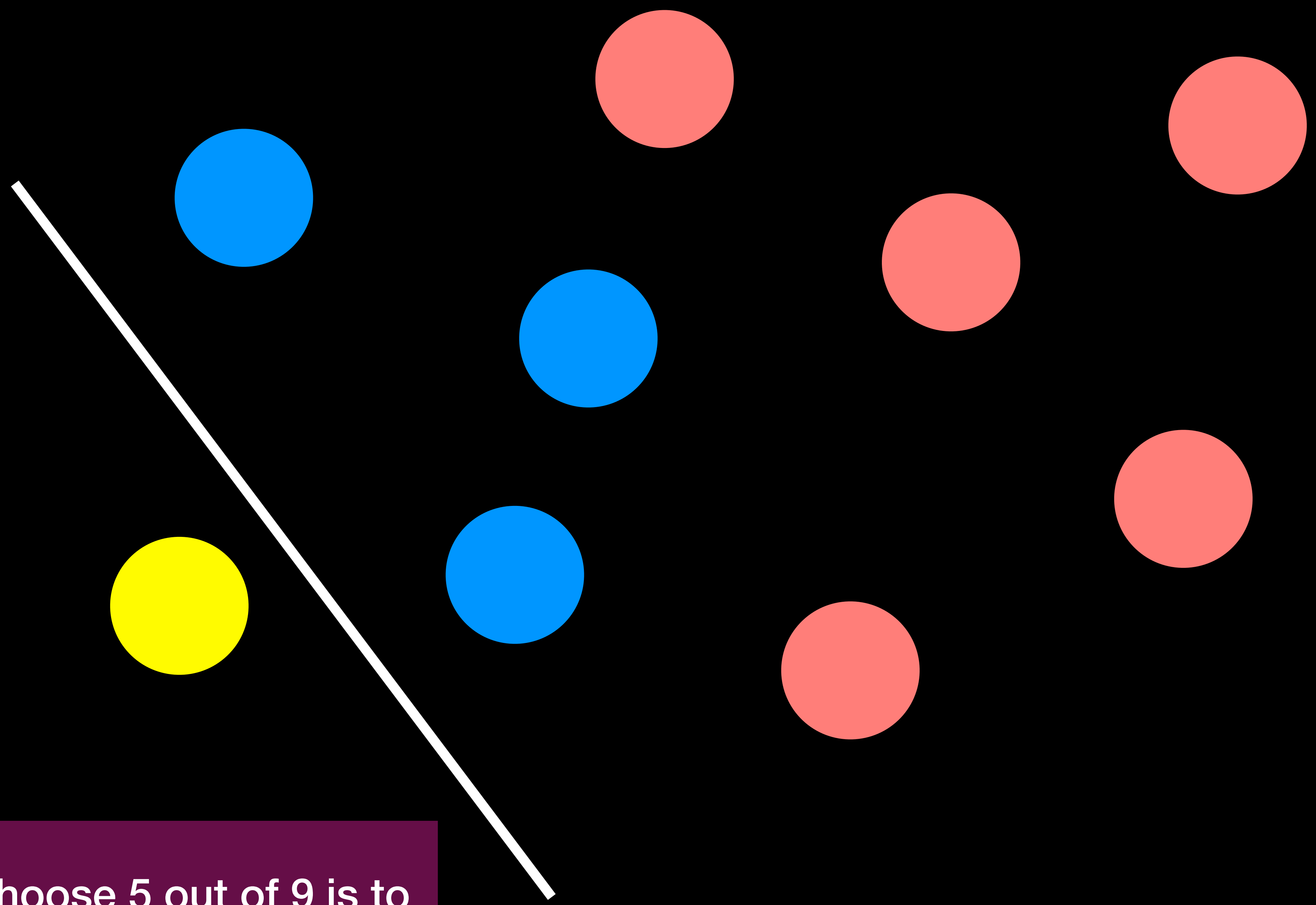
One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

Combinations (n choose k)



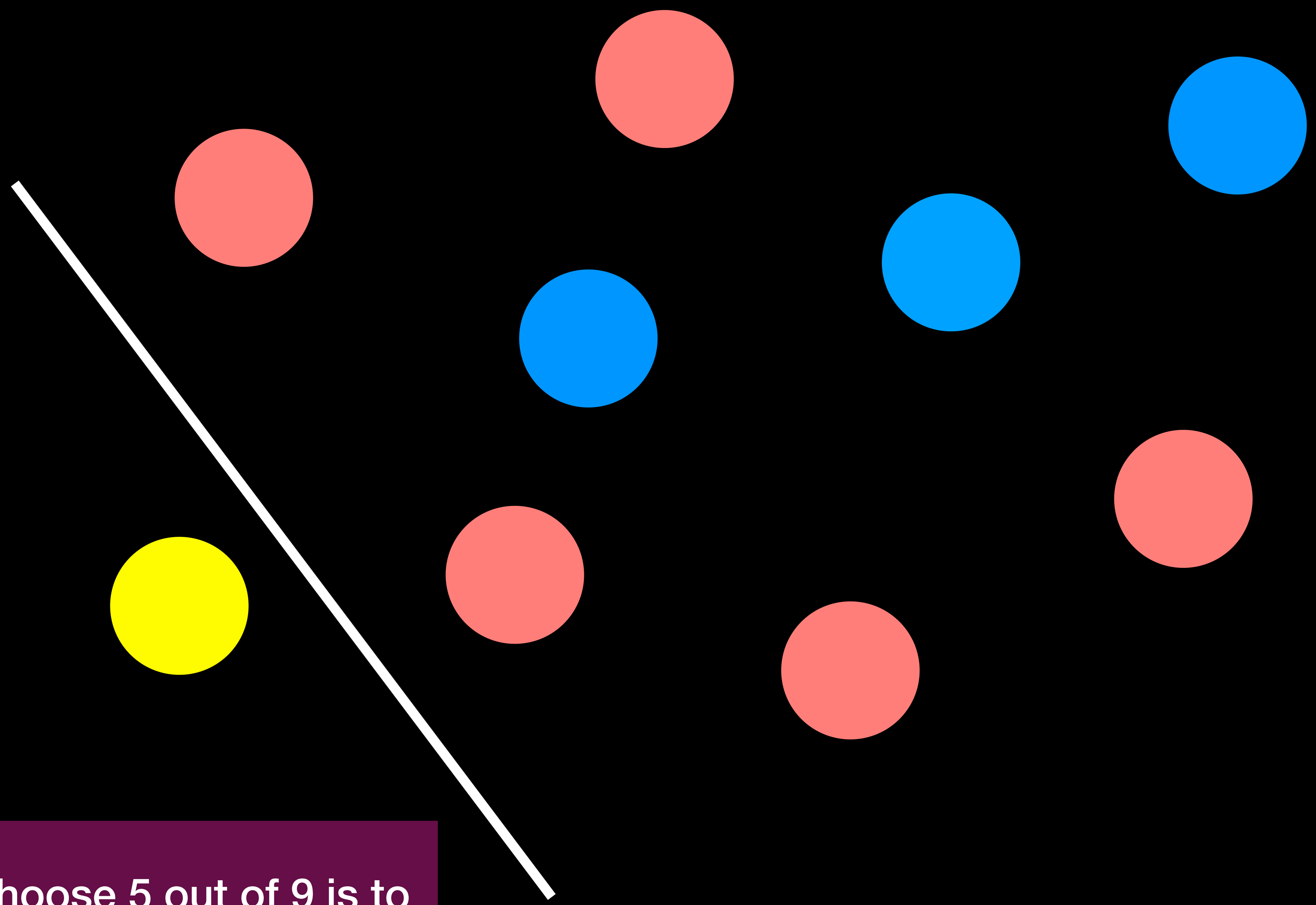
One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

Combinations (n choose k)



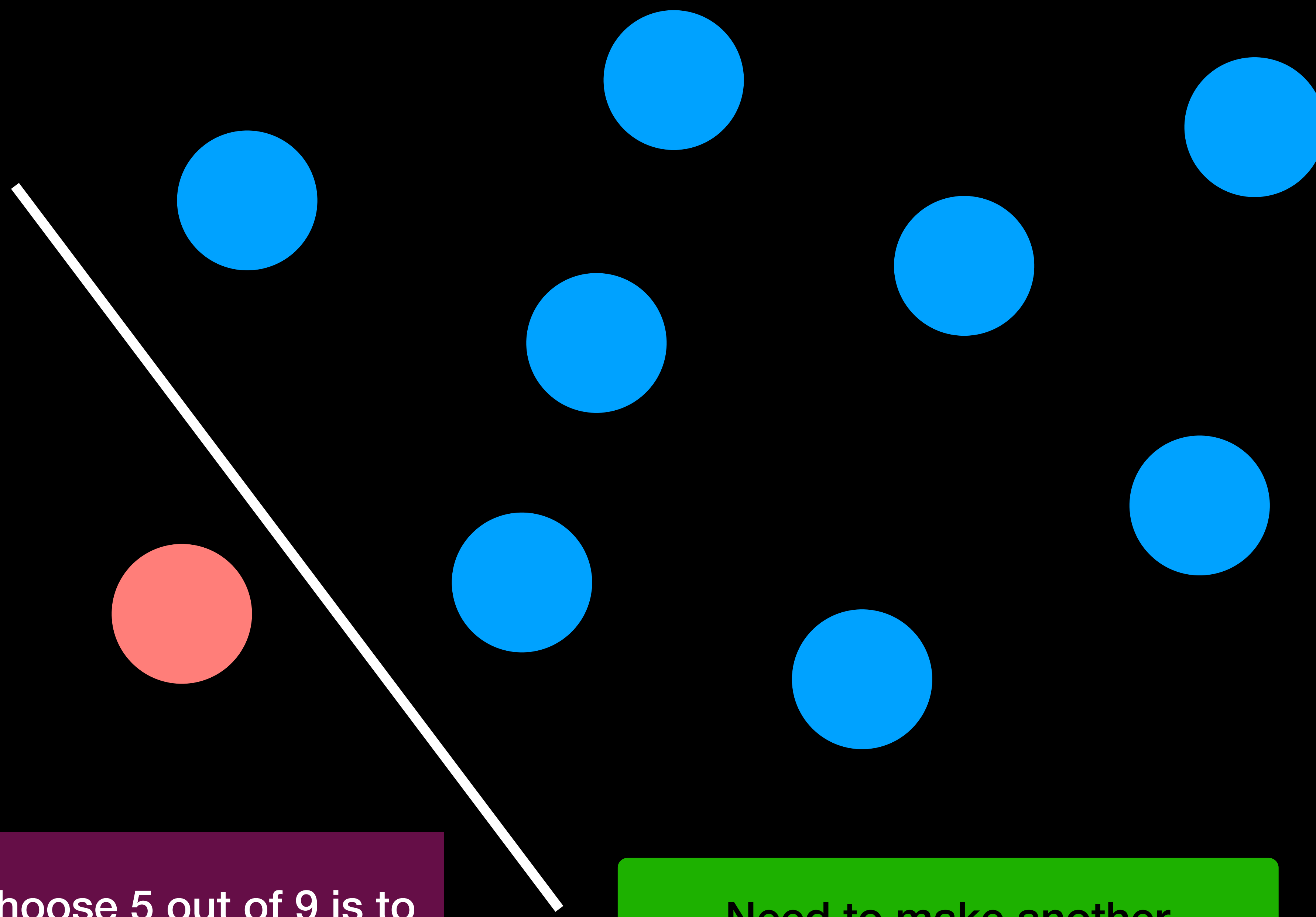
One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

Combinations (n choose k)



One way to choose 5 out of 9 is to
Exclude 1 and choose 5 out of 8

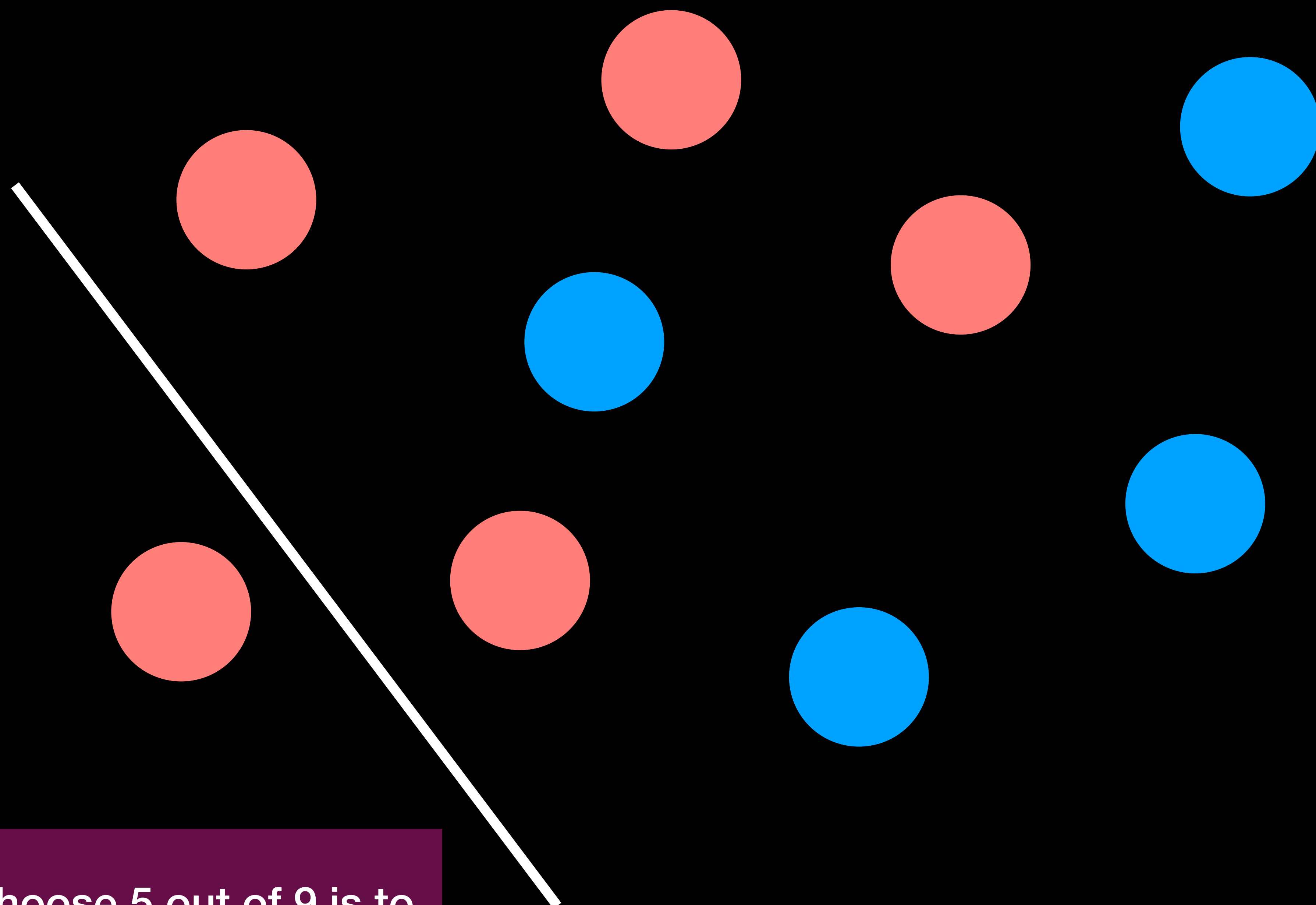
Combinations (n choose k)



One way to choose 5 out of 9 is to
Include 1 and choose 4 out of 8

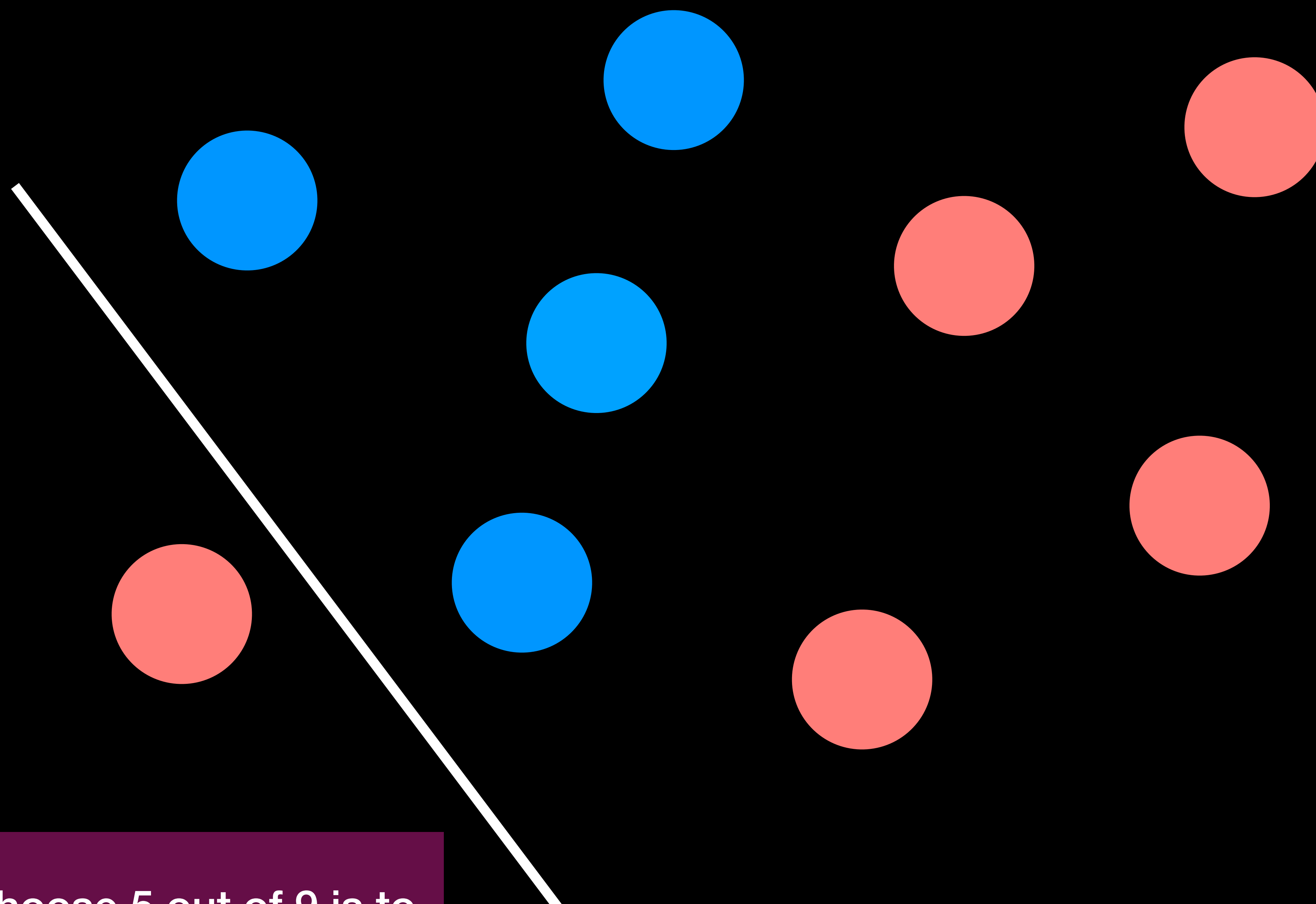
Need to make another observation

Combinations (n choose k)



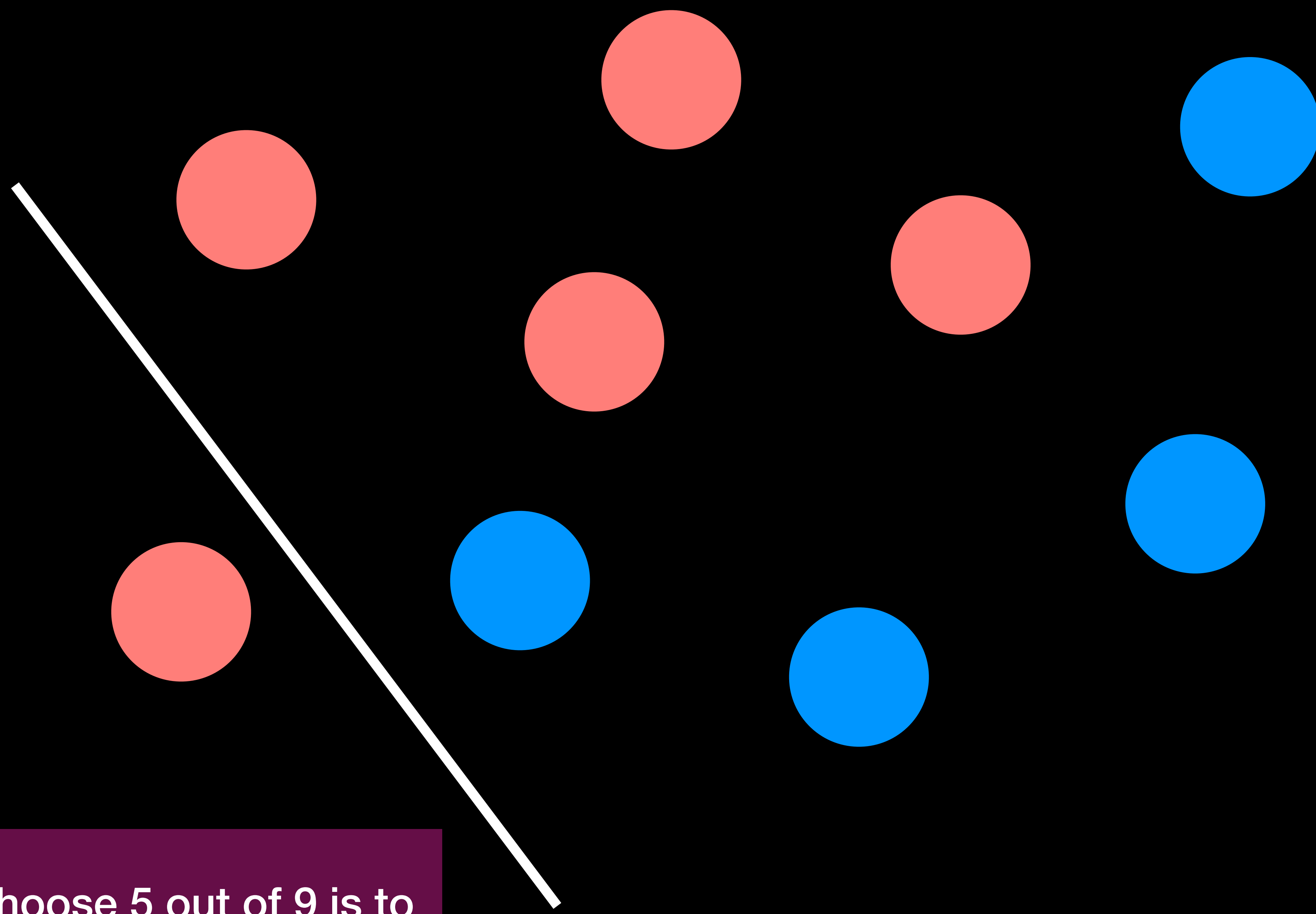
One way to choose 5 out of 9 is to
Include 1 and choose 4 out of 8

Combinations (n choose k)



One way to choose 5 out of 9 is to
Include 1 and choose 4 out of 8

Combinations (n choose k)



One way to choose 5 out of 9 is to
Include 1 and choose 4 out of 8

Count Combinations

```
int countCombinations(int n, int k)
{
    if ( (k == 0) || (k == n))
        return 1;
    else
        return countCombinations(n-1, k-1) +
                          countCombinations(n-1, k);
}
```

Recursive algorithm for computing binomial coefficients

How can you be sure it works???

You come up with an algorithm

You implement it

You test it

How can you be sure it will ALWAYS work???

How can you be sure it works???

You come up with an algorithm

You implement it

You test it

How can you be sure it will ALWAYS work???

PROVE IT!!!

Recursion and Induction

Principle of Mathematical Induction:

Suppose you want to prove that a statement `P(n)` about an integer `n` is true for every positive integer `n`.

To prove that `P(n)` is true for all $n \geq 1$, do the following two steps:

- **Base Step:** Prove that `P(1)` is true.
- **Inductive Step:** Let `k ≥ 1`. Assume `P(k)` is true, and prove that `P(k + 1)` is also true.

Recursion and Induction

```
//a: nonzero real number, n: nonnegative integer
power(a, n)
{
  if (n = 0)
    return 1
  else
    return a * power(a, n - 1)
}
```

Prove by mathematical induction on n that the algorithm above is correct. We will show P(n) is true for all $n \geq 0$, where

P(n): For all nonzero real numbers a, power(a, n) correctly computes a^n.

Recursion and Induction

Base step: If n = 0, the first step of the algorithm tells us that `power(a,0)=1`. This is correct because $a^0 = 1$ for every nonzero real number a, so P(0) is true.

Inductive step:

Let $k \geq 0$.

Inductive hypothesis: `power(a, k)` $= a^k$, for all a != 0.

We must show next that `power(a, k+1)` $= a^{k+1}$.

Since k + 1 > 0 the algorithm sets

`power(a, k + 1) = a * power(a, k)`

By inductive hypotheses `power(a, k) =` a^k

so `power(a, k + 1) = a* power(a, k)` $= a * a^k =$ $\mathbf{a^{k+1}}$

Exam Drill

Write a recursive function that returns true if the input string is a palindrome (same when reversed)

Exam Drill

Write a recursive function that returns true if the input string is a palindrome (same when reversed)

```
bool isPalindrome(std::string s)
{
    if(s.length() == 0 || s.length() == 1) //base case
        return true; //empty string or string of size 1 are palindrome
    if(s[0] == s[s.length()-1]) //if first and last char are same
        //check substring leaving out first and last character
        return isPalindrome(s.substr(1, s.length()-2));

    return false; //not palindrome
}
```

Exam Drill

Write a recursive function for the fibonacci numbers where $f(n) = f(n-1) + f(n-2)$

Exam Drill

Write a recursive function for the fibonacci numbers where f(n) = f(n-1) + f(n-2)

```
int fib(int n)
{
    if (n <= 1)//base case
        return n;
    return fib(n-1) + fib(n-2);
}
```

Exam Drill

Write a recursive function to find the max value in an array of integers

Exam Drill

Write a recursive function to find the max value in an array of integers

```
int findMax(int* a, int index) {
    if (index > 0)
        return std::max(a[index], findMax(a, index-1));
    else
        return a[0];
}
```

Exam Drill

Write a **recursive** function that finds a particular item in a **sorted** array (we will look at this algorithm after the midterm)