Recursion

– Omkar Deshpande

without further subdivision.
Reassemble the solved components to obtain the complete solution to the original problem.

"When students first encounter recursion, they often react with suspicion to the entire idea, as if they have just been exposed to some conjurer's trick, rather than a critically important programming methodology. That suspicion arises because recursion has few analogues in everyday life and requires students to think in an unfamiliar way." (Eric Roberts, *Thinking Recursively*)

Real-life counterparts of programming features

for, while loops = perform a task repeatedly (iteration)

if-then-else conditional test = make a decision based on some condition

Top-down design or Stepwise refinement via functions = Subdivide an overall high-level problem into a set of lower-level steps

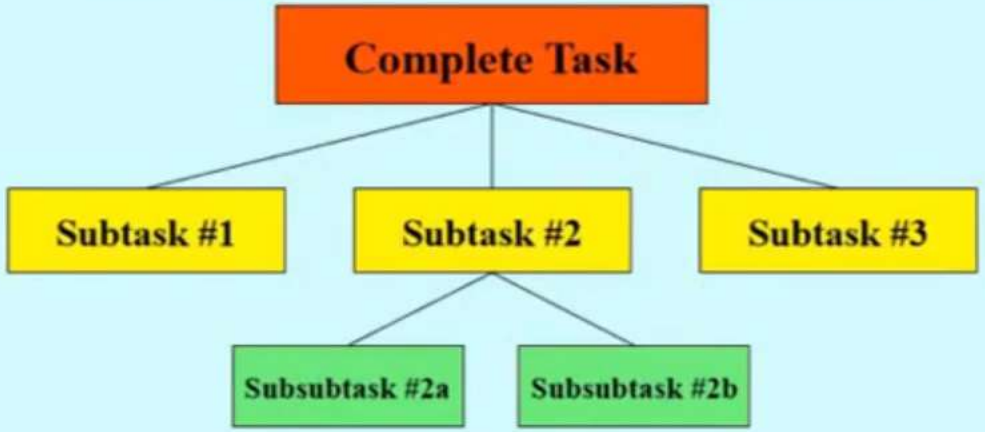
Bake a cake:

1. Get cookbook
2. Get All Ingredients
3. Read & follow instructions
   - Mix flour, milk, sugar etc
   - Set oven to a particular temp, etc

- Set oven to a particular temp, etc

Stepwise Refinement

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called ***stepwise refinement*** or ***decomposition***.



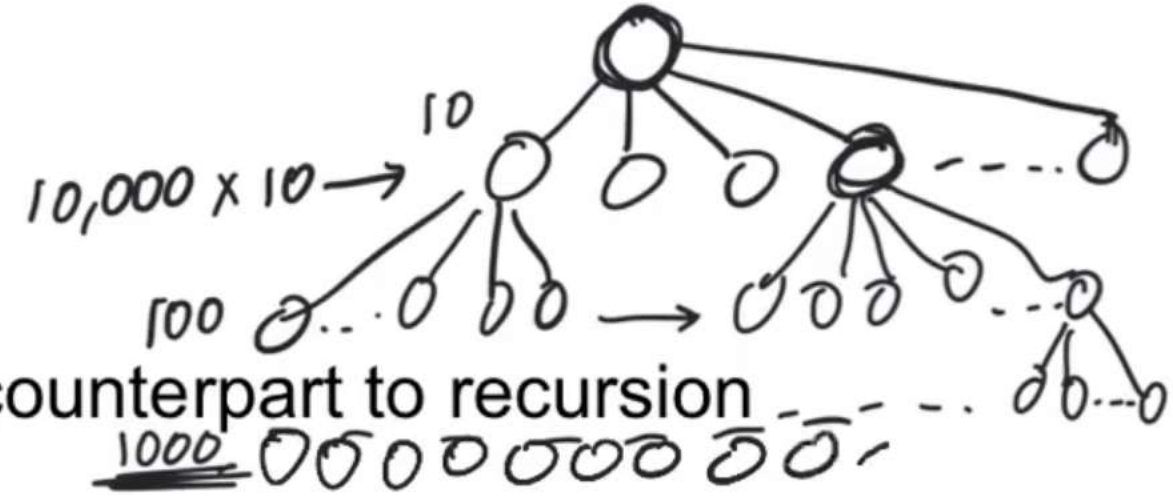
Recursion = Solve large problems by reducing them to smaller problems **of the same form**.

Reduce a large problem to one or more subproblems that are
1) Identical in structure to the original problem
2) Somewhat simpler to solve

Then use the same decomposition technique to divide the subproblems into new ones that are even simpler… until they become so simple that you can solve them without further subdivision.
Reassemble the solved components to obtain the complete solution to the original problem.

Then use the same decomposition technique to divide the subproblems into new ones that are even simpler… until they become so simple that you can solve them without further subdivision.
Reassemble the solved components to obtain the complete solution to the original problem.

"When students first encounter recursion, they often react with suspicion to the entire idea, as if they have just been exposed to some conjurer's trick, rather than a critically important programming methodology. That suspicion arises because recursion has few analogues in everyday life and requires students to think in an unfamiliar way." (Eric Roberts, *Thinking Recursively*)



A crude real-life counterpart to recursion

You have been appointed as the growth manager for a non-profit company. Your job is to raise 100000$.

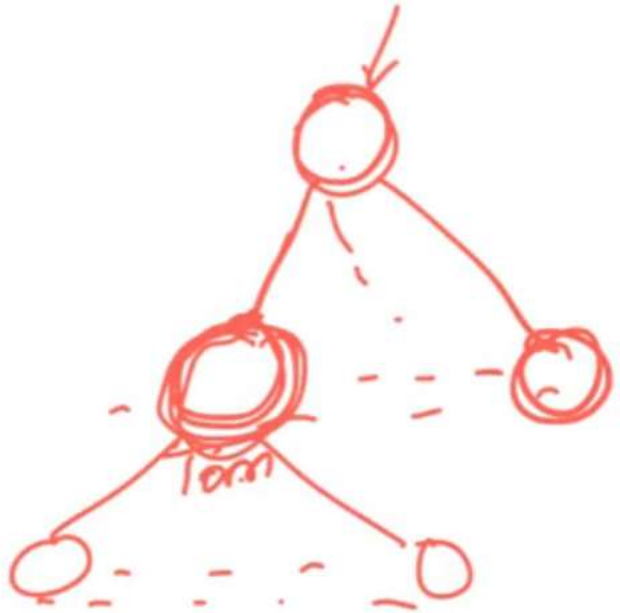
This task greatly exceeds your own capacity. So you need to delegate the work to others.

You find 10 volunteers, and give them each the task of raising 10000$ each.

Each of them finds 10 volunteers, each of who is tasked with raising 1000$. They in turn could find volunteers who only need to raise only 100$.

in turn could find volunteers who only need to raise only 100$.

Pseudocode for strategy



```
function raiseMoney (int n):
    if n <= 100:
        collect the money from a single donor
    else:
        find 10 volunteers
        get each of them to collect (n/10) dollars
        combine the money raised by the volunteers
```

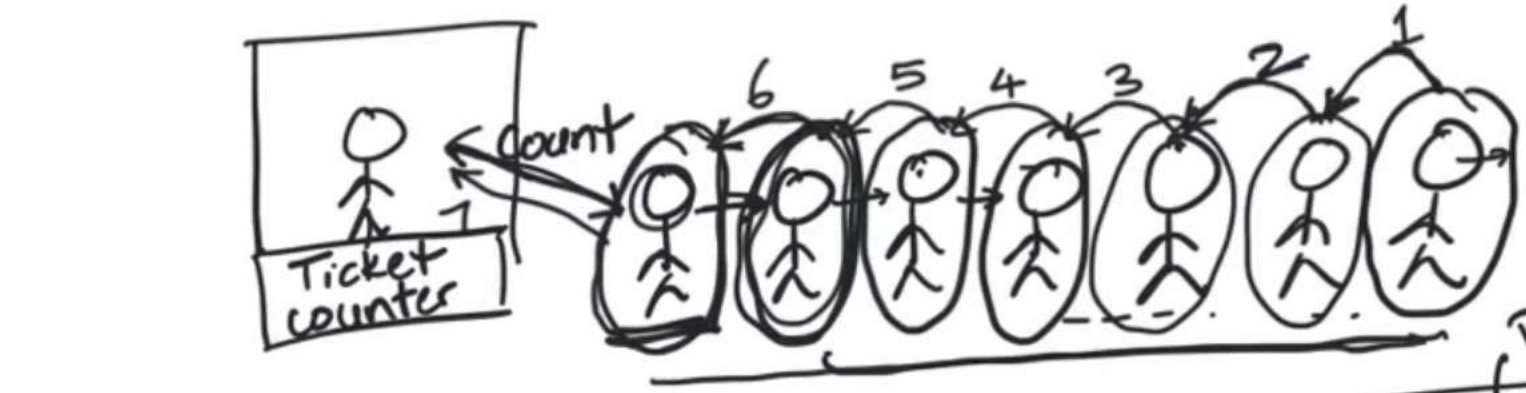
This has the same form as the original problem, and the problem size is smaller.

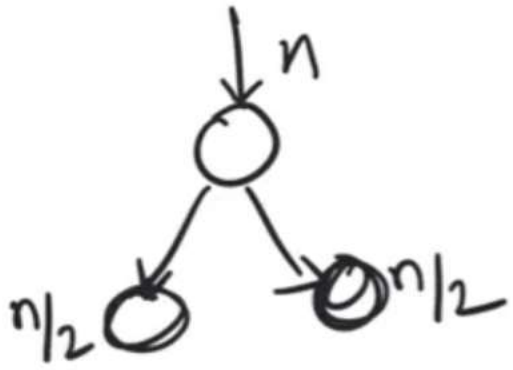
Pseudocode for strategy

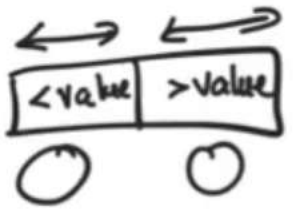
```
function raiseMoney (int n):
    if n <= 100:
        collect the money from a single donor
    else:
        find 10 volunteers
        for each volunteer: call raiseMoney(n/10)
        combine the money raised by the volunteers
```

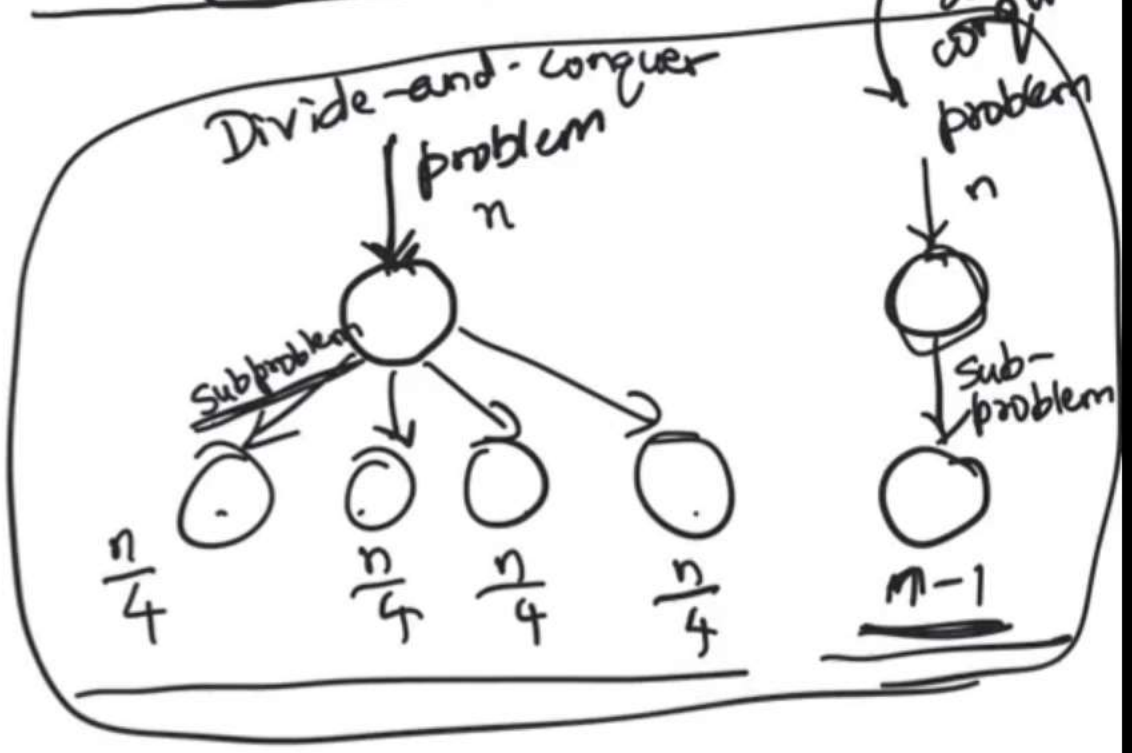
Pseudocode for general recursive solution

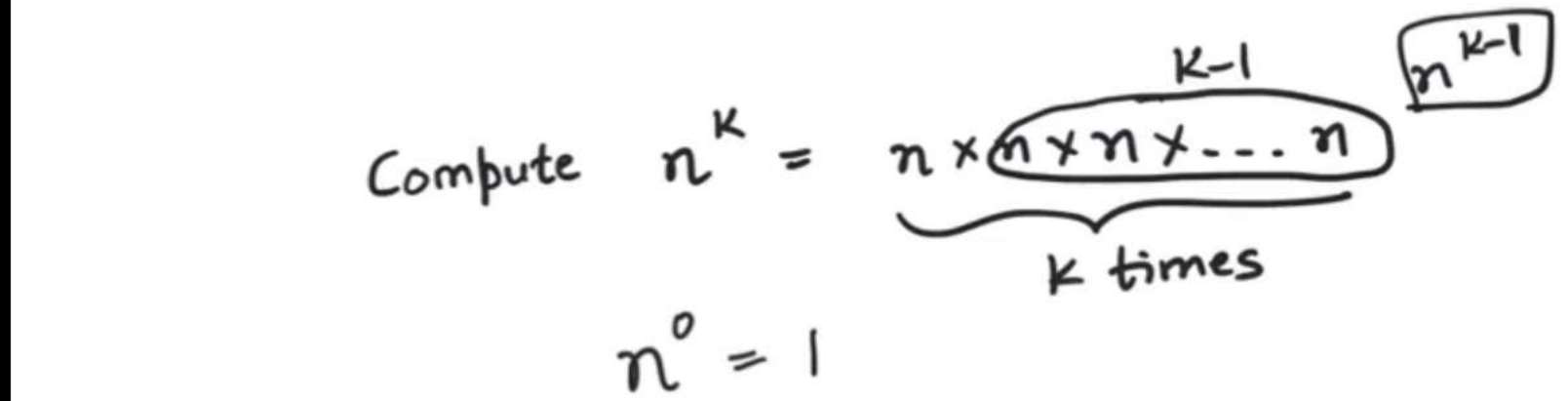
```
if (test for a simple case):
    compute a simple solution without using recursion
else:
    #Divide-and-conquer or Decrease-and-conquer
    break the problem into subproblems of the same form
    solve each of the subproblems by calling this function recursively
    reassemble the subproblem solutions into a solution for the whole
```


count
6
5
4
3
2
1
Ticket counter


n
n/2
n/2


<value
>value


Divide-and-conquer
problem
n
Subproblem
n/4
n/4
n/4
n/4
Decrease and conquer
problem
n
Sub-problem
n-1



Write a recursive implementation of a function that raises a number to a power.

```
int RaiseIntToPower(int n, int k)
```

Use this mathematical definition:

$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

Write a recursive implementation of a function that raises a number to a power.

```
int RaiseIntToPower(int n, int k)
```

Use this mathematical definition:

$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

```
def RaiseIntToPower(n, k):
    if k == 0:
        return 1
    else:
        return n * RaiseIntToPower(n, k-1)
```

```
def RaiseIntToPower(n,k):
    result = 1
    for i in 1 to k:
        result = result * n
    return result
```

n^k

$1 \rightarrow 2$

$2 \rightarrow 3$

$\vdots$

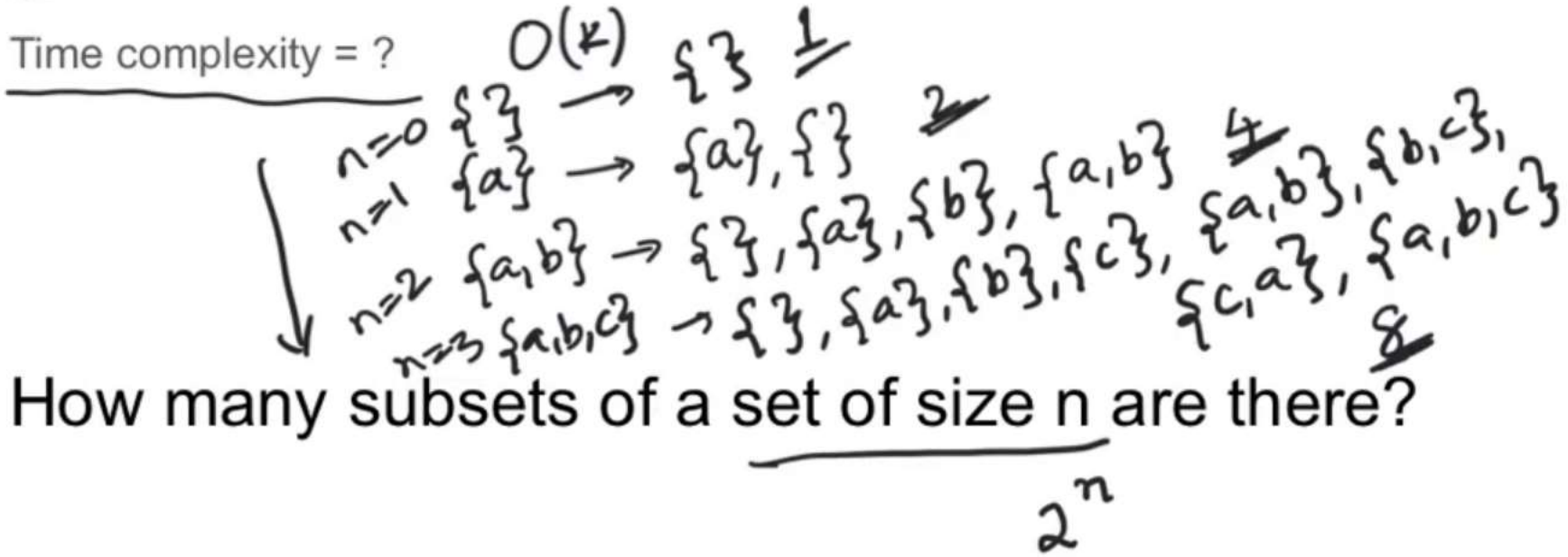
$k-1 \rightarrow k$

Again, a decrease-and-conquer strategy like this (problem of size n constructed from solution to subproblem of size n-1) means it can be easily implemented iteratively as well.

Time complexity = ?

iteratively as well.

Time complexity = ?

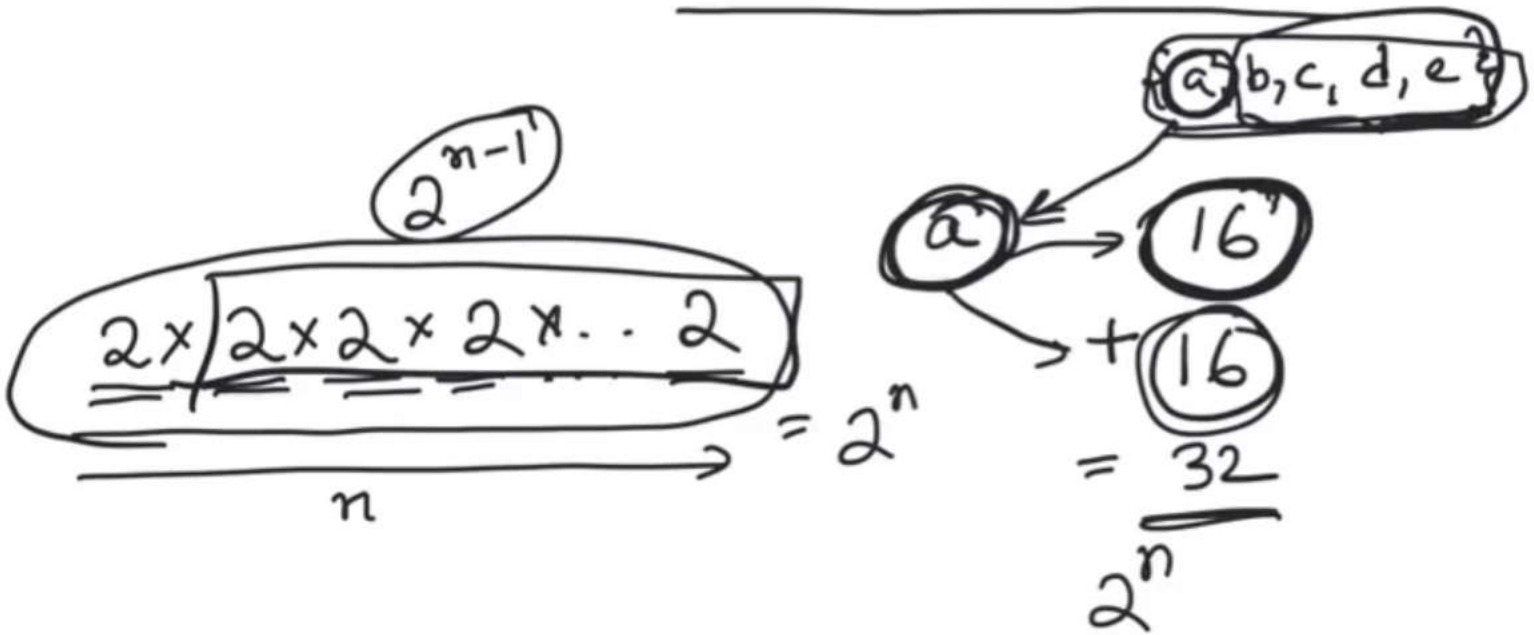


How many subsets of a set of size n are there?

2^n

How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?



How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?

S(n) = Number of subsets of size n

S(n) = S(n-1) + S(n-1) = 2S(n-1)

Base case: S(0) = 1

How many subsets of a set of size n are there?

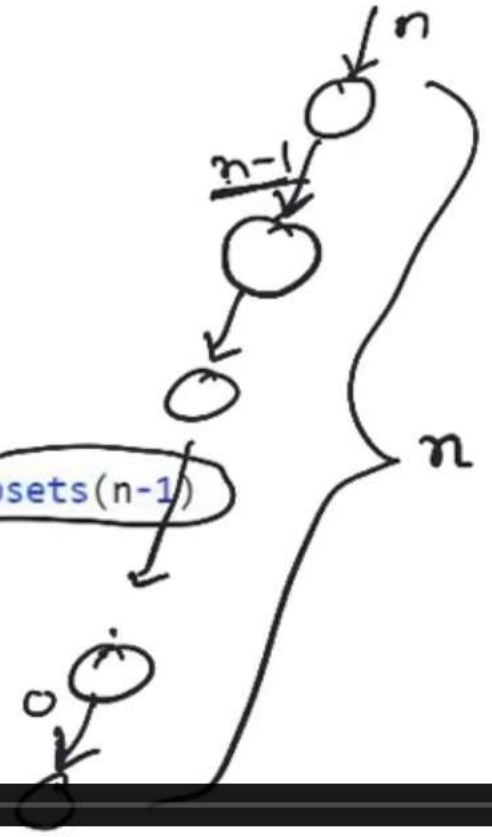
Hint: What if I knew how many subsets of size n-1 are there?

S(n) = Number of subsets of size n

S(n) = S(n-1) + S(n-1) = 2S(n-1)

Base case: S(0) = 1

```
def subsets(n):
    if n == 0:
        return 1
    else:
        return 2*subsets(n-1)
```



Time complexity = ?

What if we wrote the code like this?

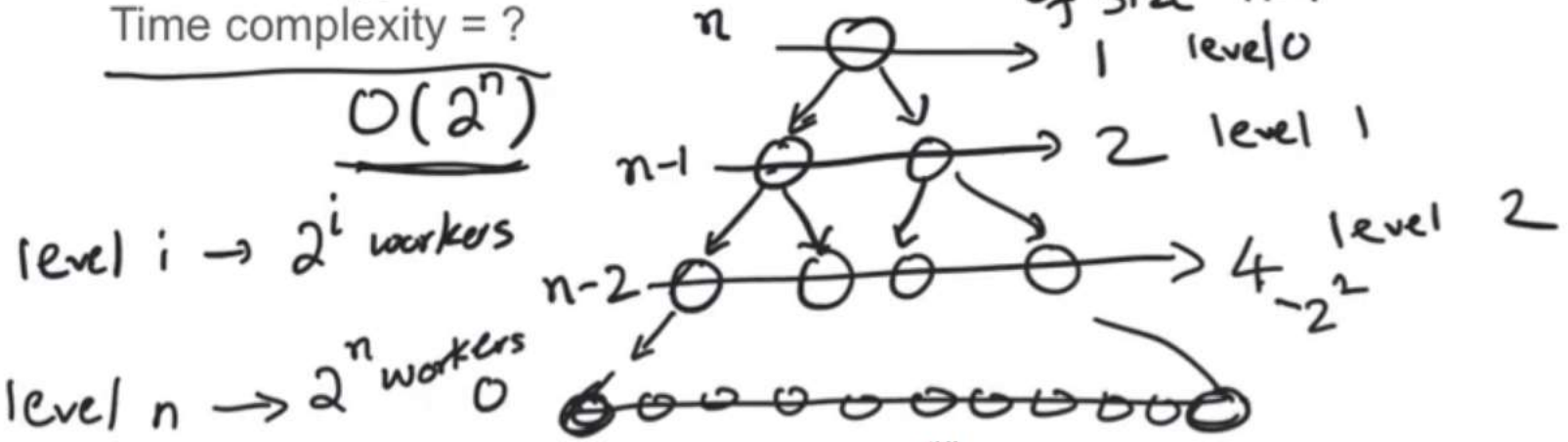
```
def subsets(n):
    if n == 0:
        return 1
    else:
        return subsets(n-1) + subsets(n-1)
```

$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$

2 subproblems of size n-1

Time complexity = ?

$O(2^n)$



$2^n = 2 \times 2^{n-1}$

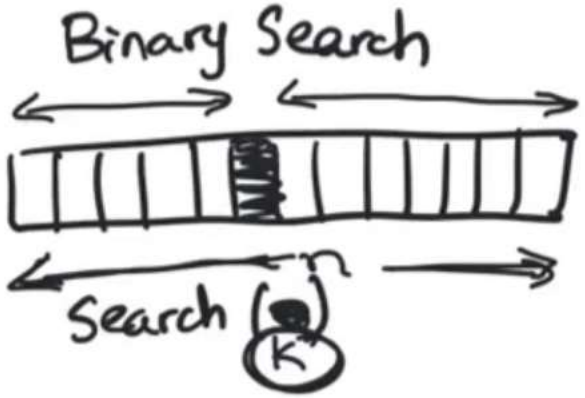
Decrease-and-conquer

$n \to n-1$

$n \to n/2$

$2^{n/2} \times 2^{n/2}$

$= (2^{n/2})^2$



2^{20}

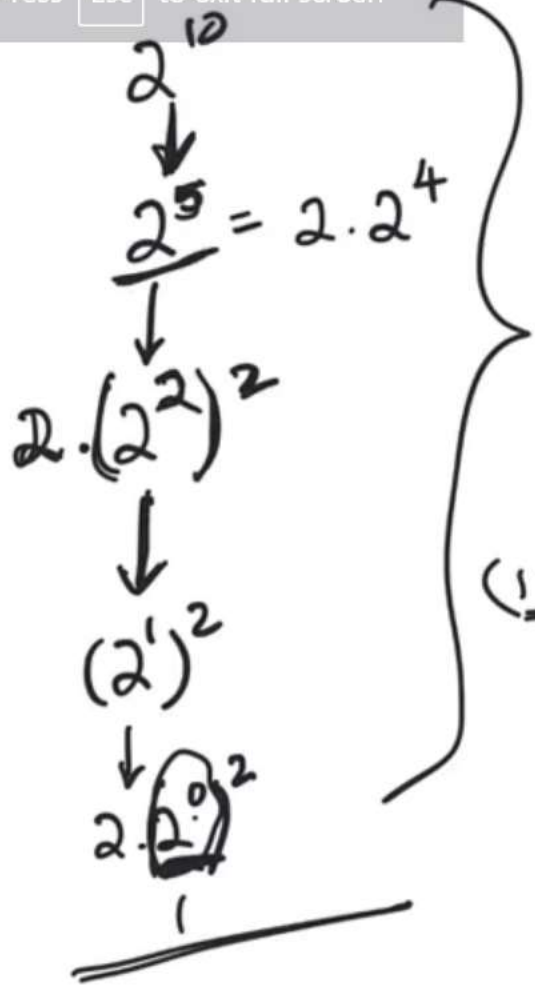
2^{10}

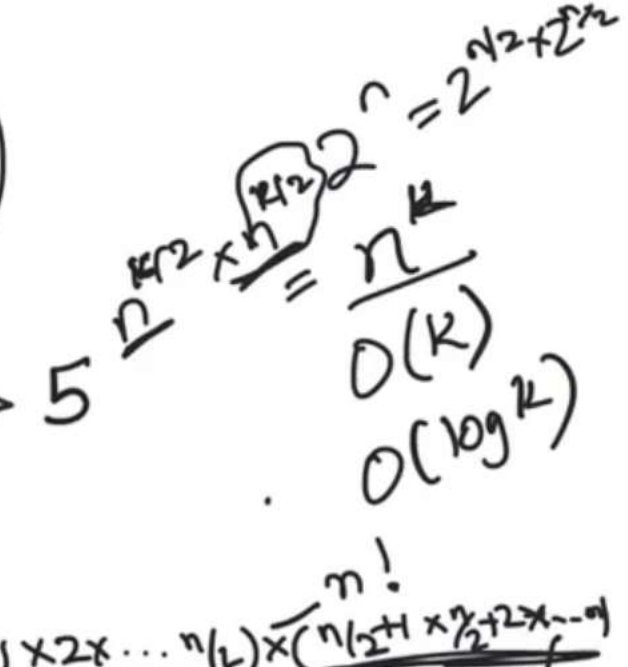
2^5

$n \to n/2$

$2^{95} = 2 \cdot 2^{94}$

$(2^{47})^2$





$$(1 \times 2 \times \cdots n/2) \times (n/2+1 \times n/2+2 \times \cdots n)$$

$n!$

$$T(n) = T(n/2) + 1$$

$$= T(n/4) + 1 + 1$$

$$= T\left(\frac{n}{8}\right) + 1 + 1 + 1$$

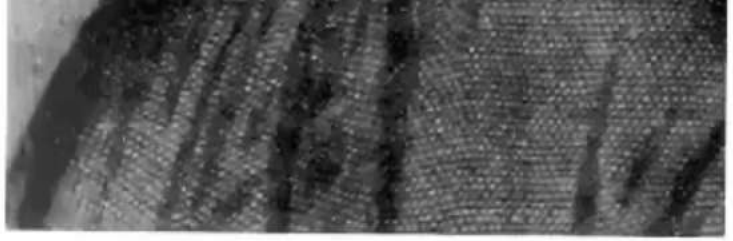
$$= T\left(\frac{n}{2^i}\right) + i$$

$$\frac{n}{2^i} = 1 \Rightarrow i \sim \log_2 n$$

$$= T(1) + \log_2 n$$

$$= O(\log n)$$

Leonardo of Pisa
(1170-1250)



Pisa

Location of Pisa in Italy

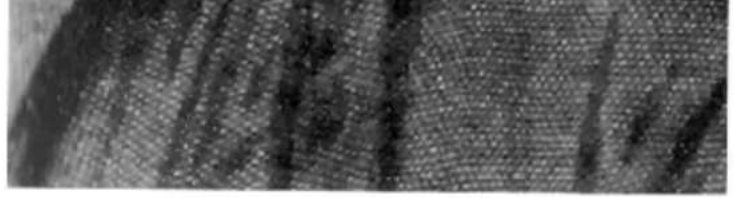
The leaning tower of Pisa

Location of Pisa in Italy

The leaning tower of Pisa

Leonardo of Pisa
(1170-1250)

filius Bonaccio, “son of Bonaccio”: Fibonacci



- Travelled with his father to Algeria as a young boy.
- Learnt about the Hindu-Arabic numeral system there

00:47

- Fibonacci recognized the advantages of using the Hindu-Arabic numeral system (with a positional notation and zero symbol) over the clumsy Roman system still used in Italy.
- He returned to Pisa in 1202 and wrote a book explaining the virtues of this number system “in order that the Latin race may no longer be deficient in that knowledge.”

- He returned to Pisa in 1202 and wrote a book explaining the virtues of this number system "in order that the Latin race may no longer be deficient in that knowledge."

Liber abaci (Book of Counting)

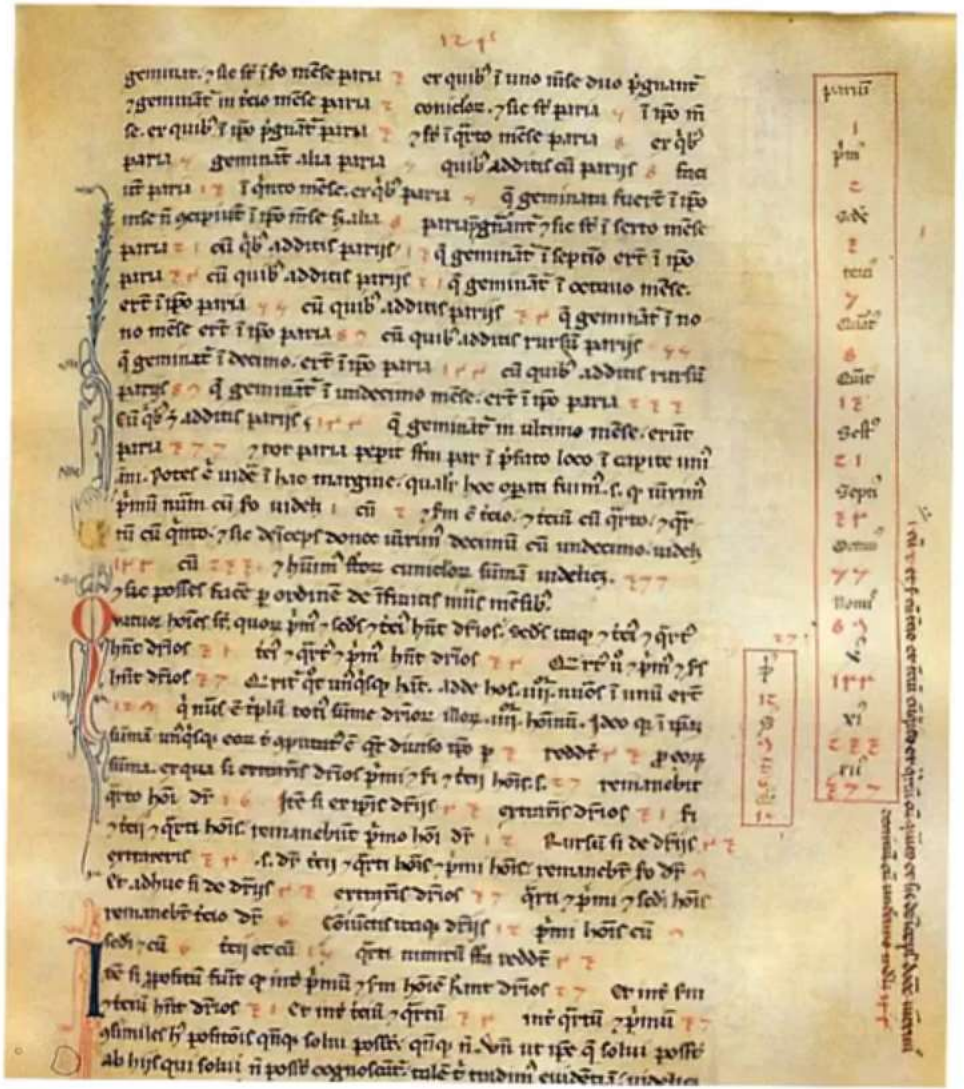
Chapter 1:

"These are the nine figures of the Indians:

9 8 7 6 5 4 3 2 1.

With these nine figures, and with this sign 0… any number may be written, as will be demonstrated below."

demonstrated below."



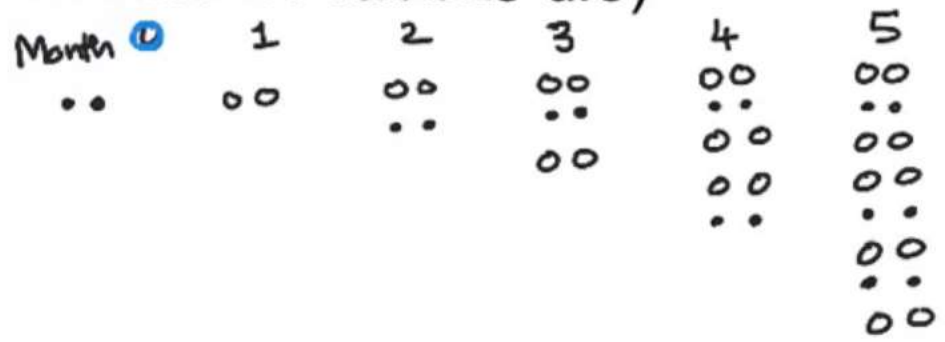
"It is ironic that Leonardo, who made valuable contributions to mathematics, is remembered today mainly because a 19th-century French number theorist, Édouard Lucas... attached the name Fibonacci to a number sequence that appears in a trivial problem in Liber abaci"

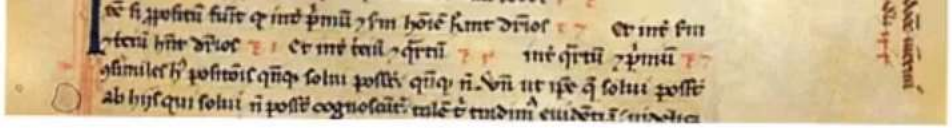
(Martin Gardner, Mathematical Circus)

Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)

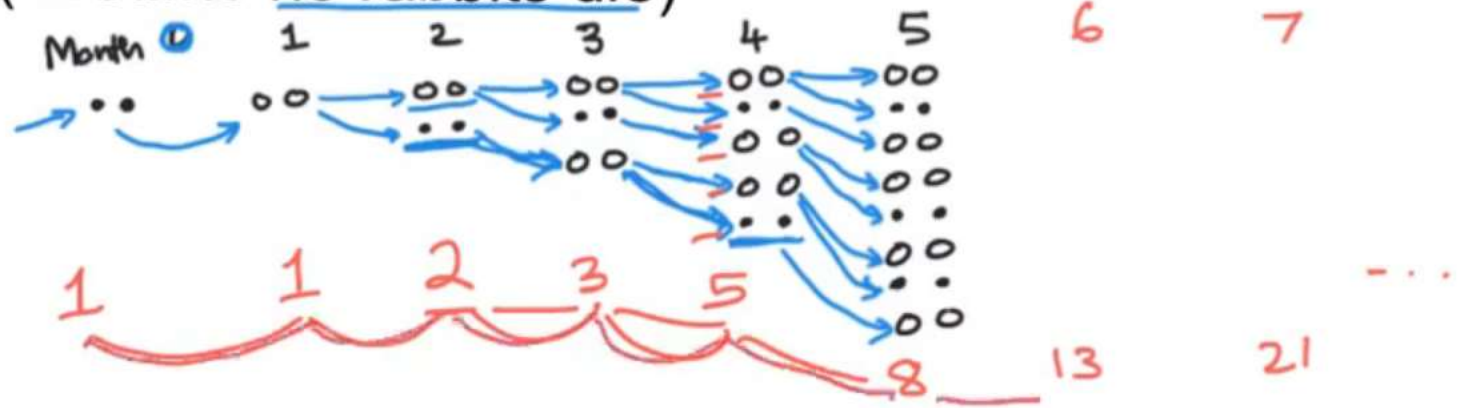




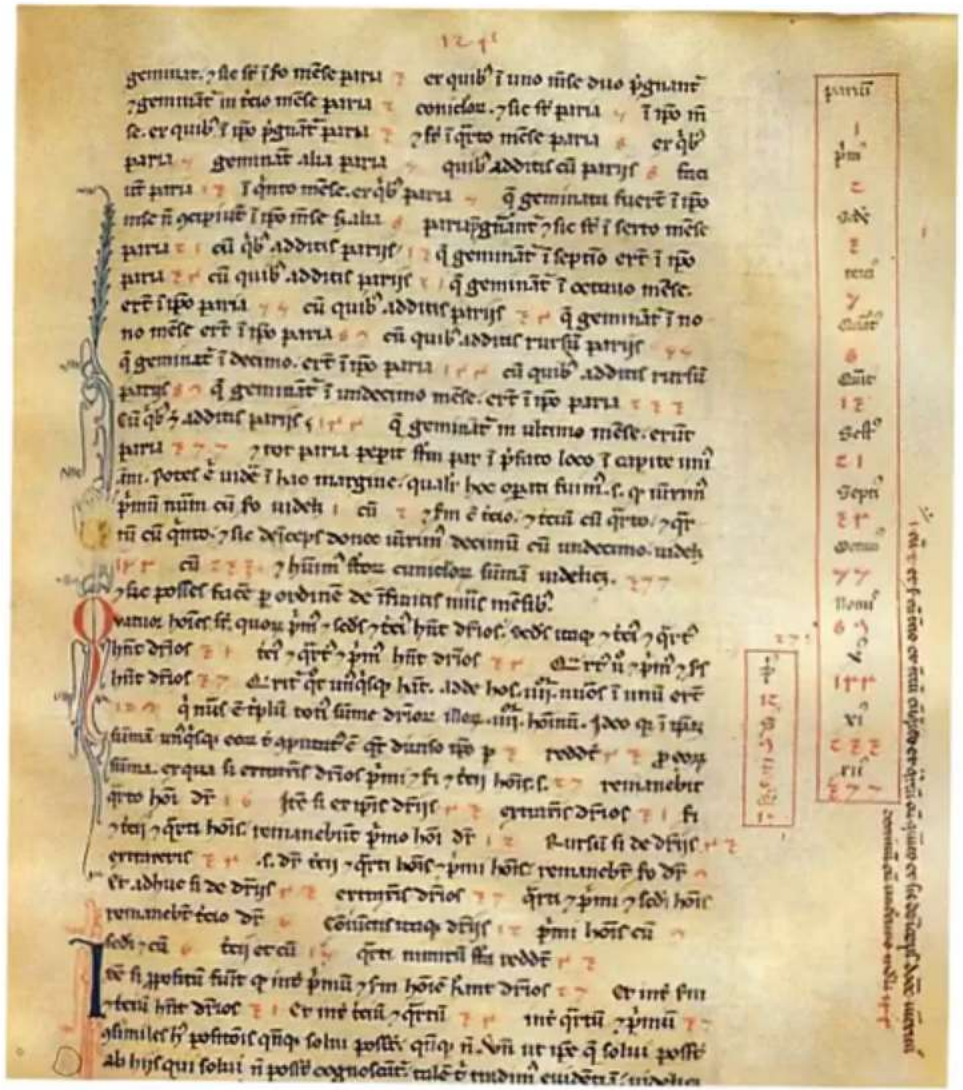
Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)

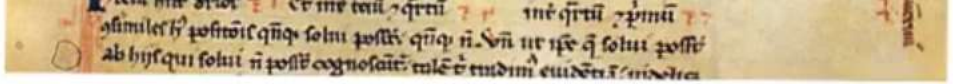


demonstrated below."



"It is ironic that Leonardo, who made valuable contributions to mathematics, is remembered today mainly because a 19th-century French number theorist, Édouard Lucas... attached the name Fibonacci to a number sequence that appears in a trivial problem in Liber abaci"

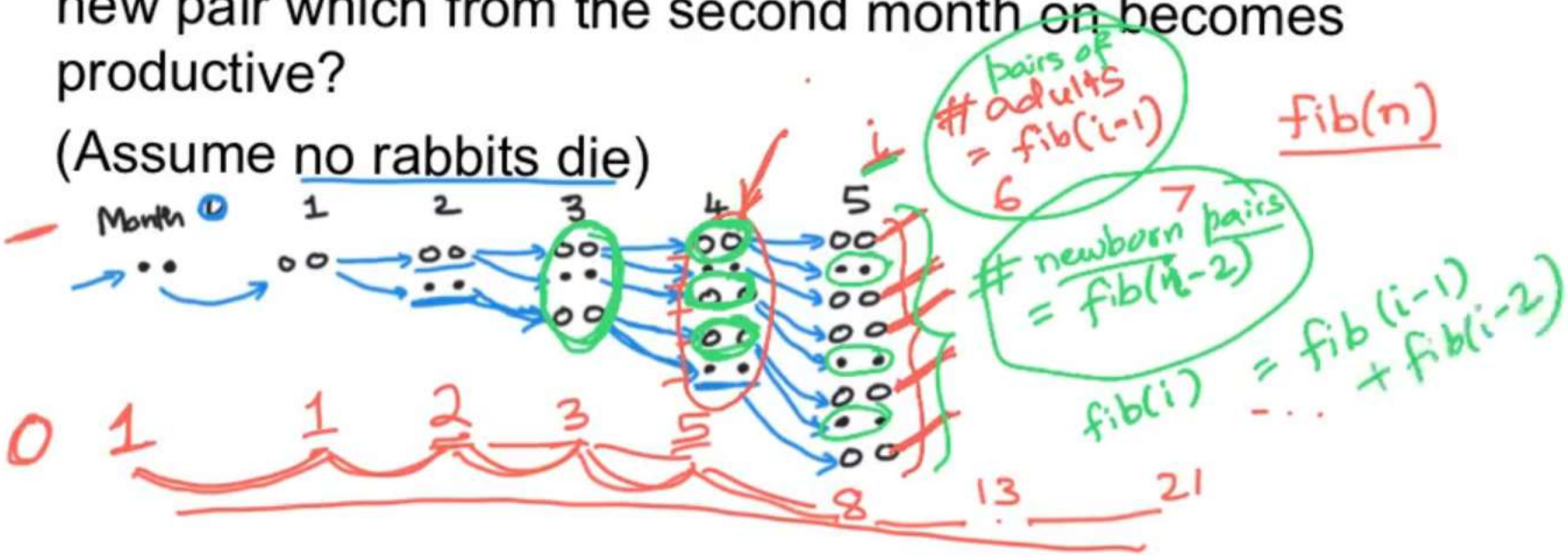
(Martin Gardner, Mathematical Circus)

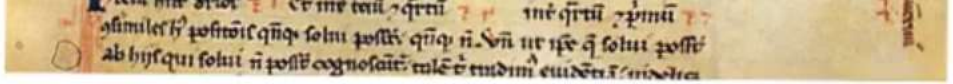


Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)

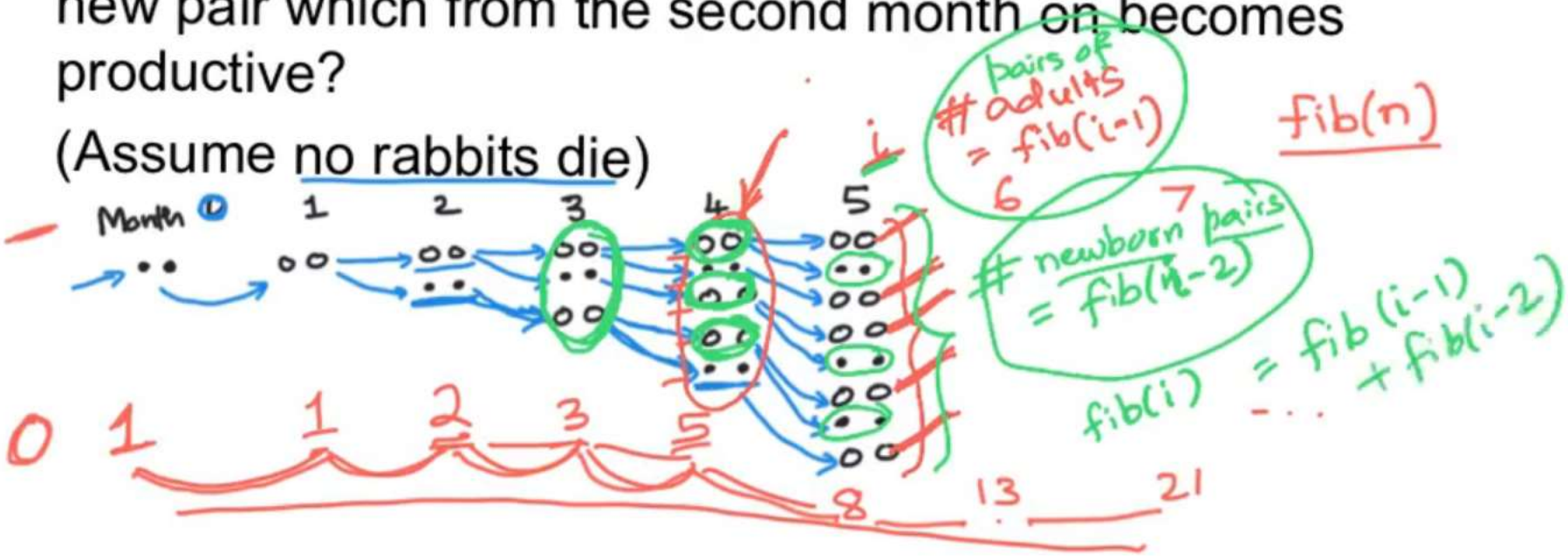


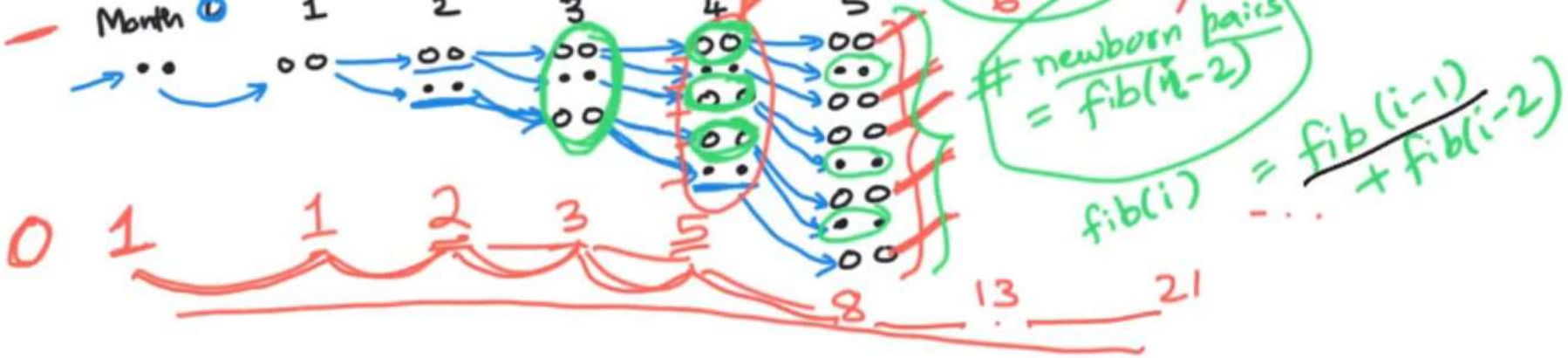


Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)



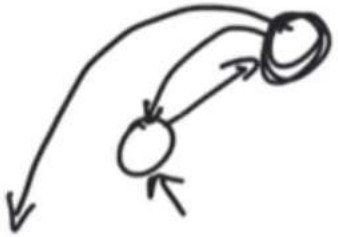


New month's adults = All the rabbit pairs from previous month = fib

New month's young = Previous month's adults = Previous to previous month's rabbit pairs = fib(i-2)

= fib(i-2)

Fibonacci sequence



fib(n) = fib(n-1) + fib(n-2)

Need to specify two starting terms (base cases):

fib(1) = 1

fib(0) = 0

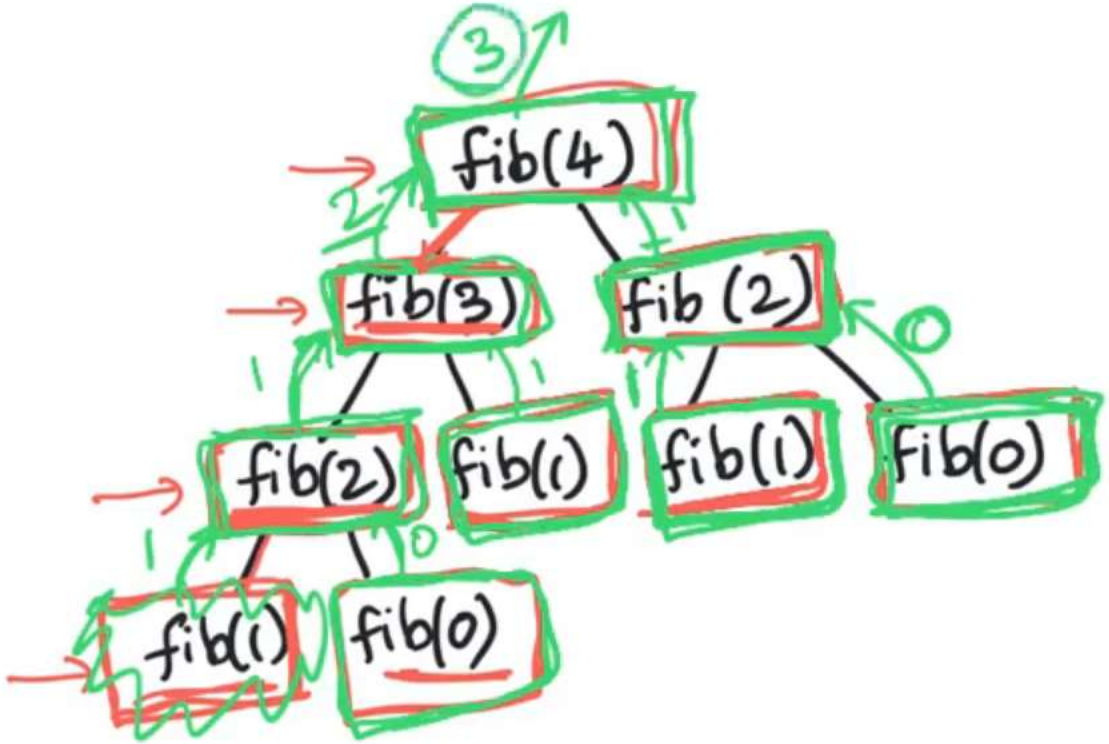
fib(2) = fib(1) + fib(0) = 1 + 0 = 1

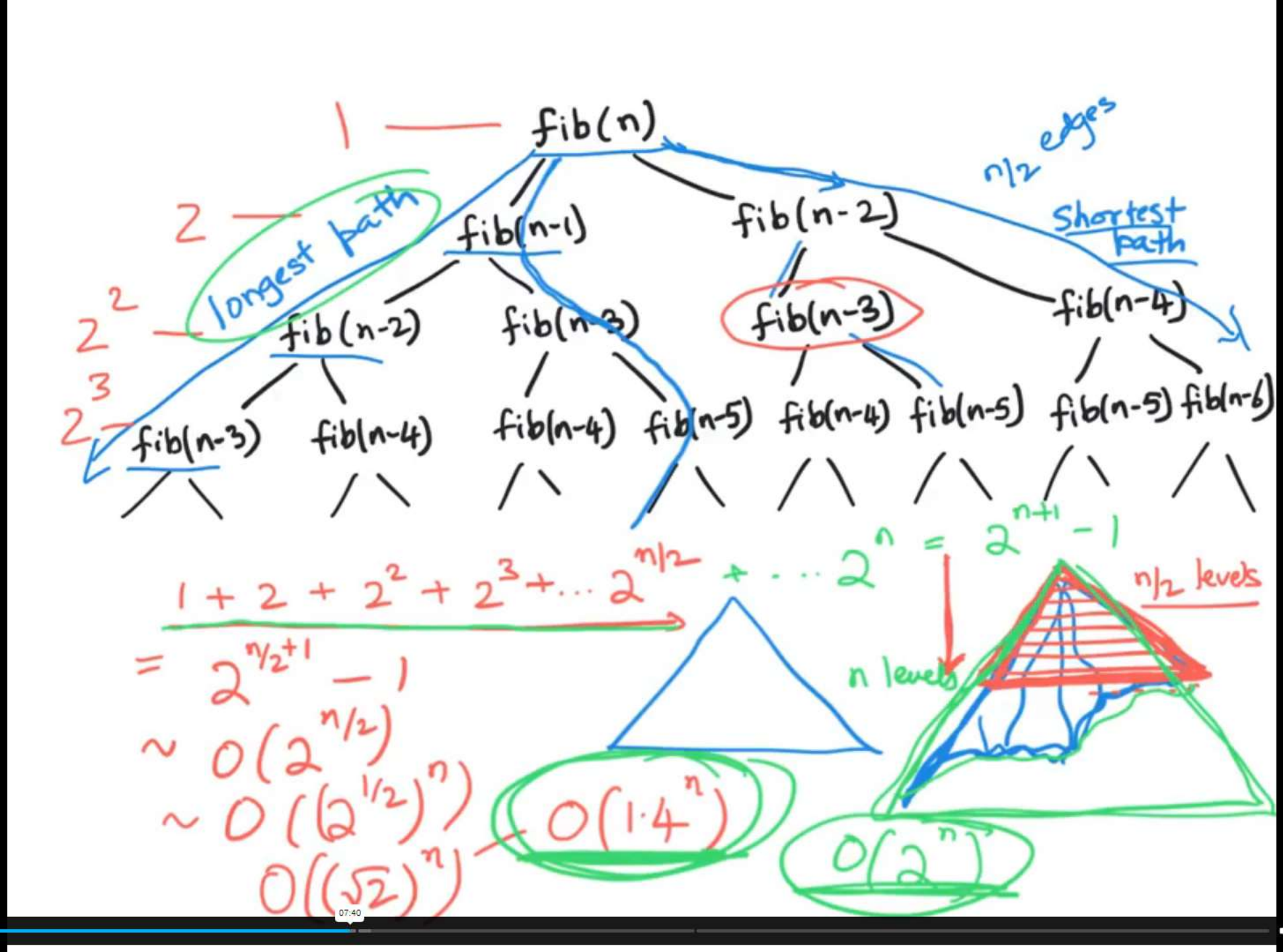
fib(3) = fib(2) + fib(1) = 1 + 1 = 2

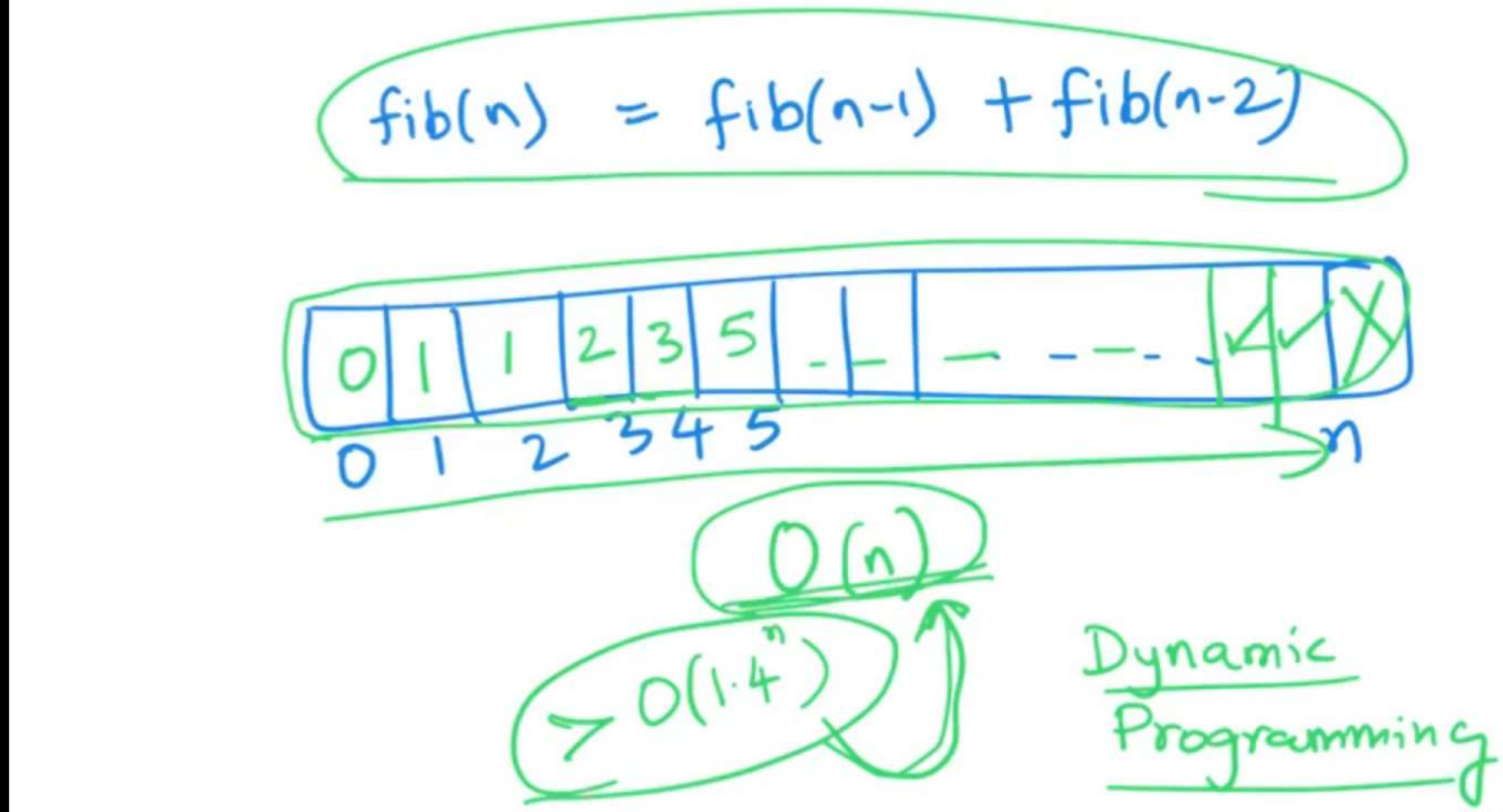
Recursive implementation of Fibonacci function fib(n)

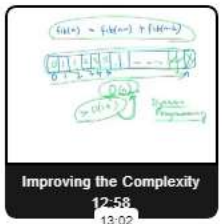
```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Time complexity = ?

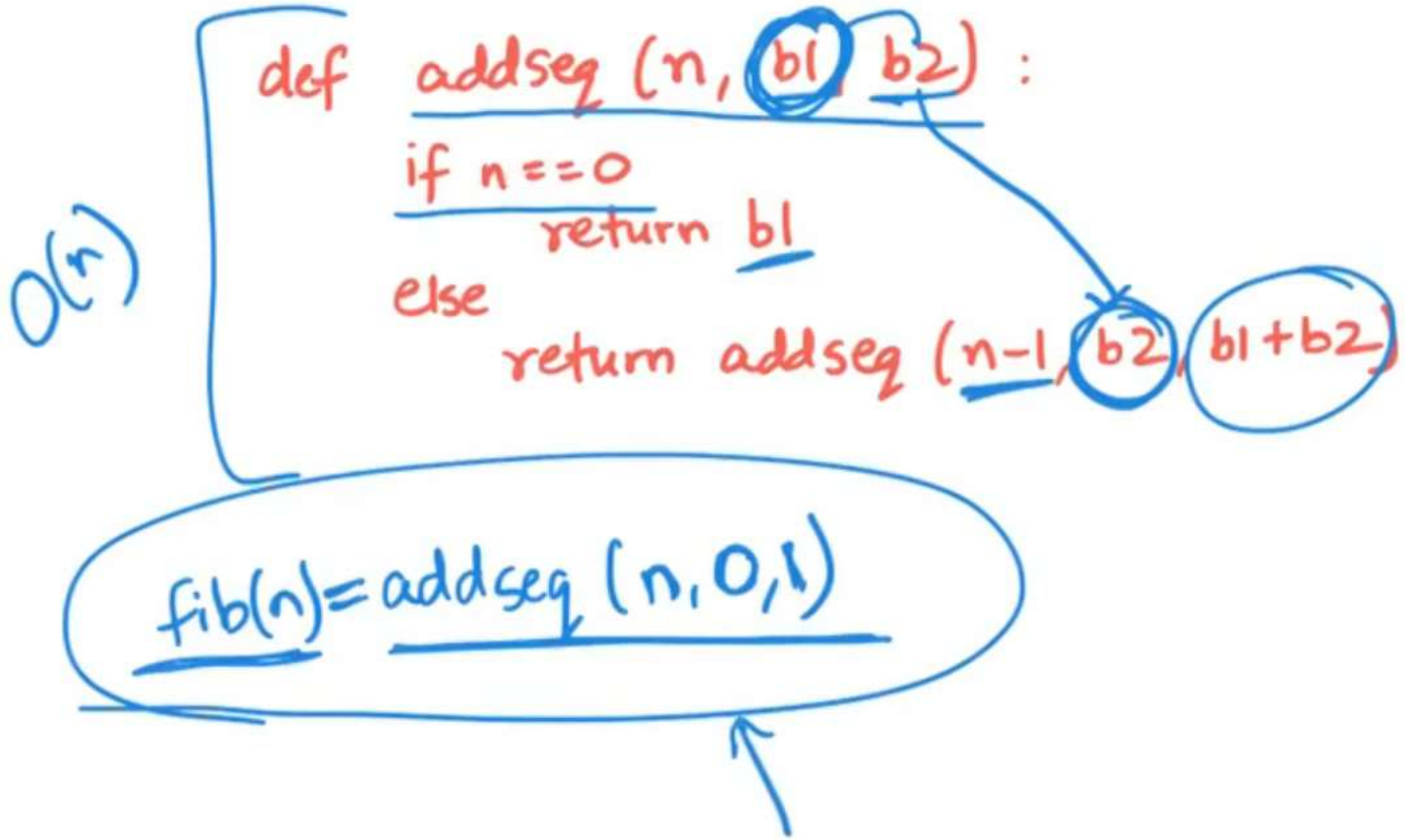


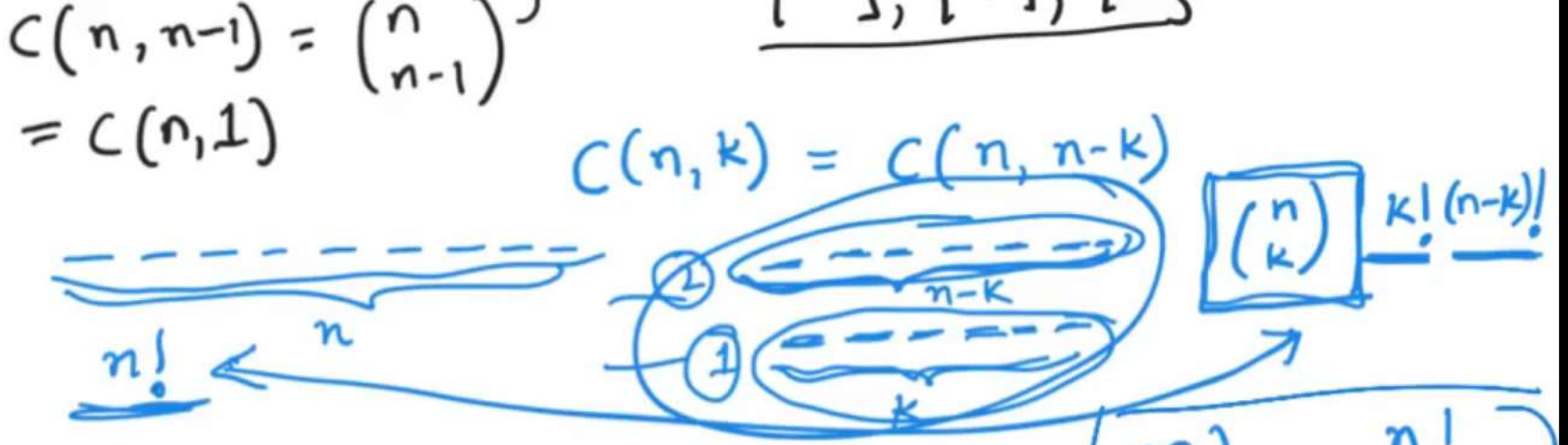

1
fib(n)
2
longest path
fib(n-1)
fib(n-2)
n/2 edges
shortest path
2^2
fib(n-2)
fib(n-3)
fib(n-3)
fib(n-4)
2^3
fib(n-3)
fib(n-4)
fib(n-4)
fib(n-5)
fib(n-4)
fib(n-5)
fib(n-5)
fib(n-6)
1 + 2 + 2^2 + 2^3 + ... 2^{n/2} + ... 2^n = 2^{n+1} - 1
= 2^{n/2+1} - 1
~ O(2^{n/2})
~ O((2^{1/2})^n)
O((√2)^n)
O(1.4^n)
n levels
n/2 levels
O(2^n)


fib(n) = fib(n-1) + fib(n-2)
0 1 1 2 3 5
0 1 2 3 4 5 n
O(n)
> O(1.4^n)
Dynamic Programming


Improving the Complexity
12:58

13:02


def addseq (n, b1, b2) :
if n==0
return b1
else
return addseq (n-1, b2, b1+b2)
O(n)
fib(n)=addseq (n,0,1)



C(n,k) - "Combinations"

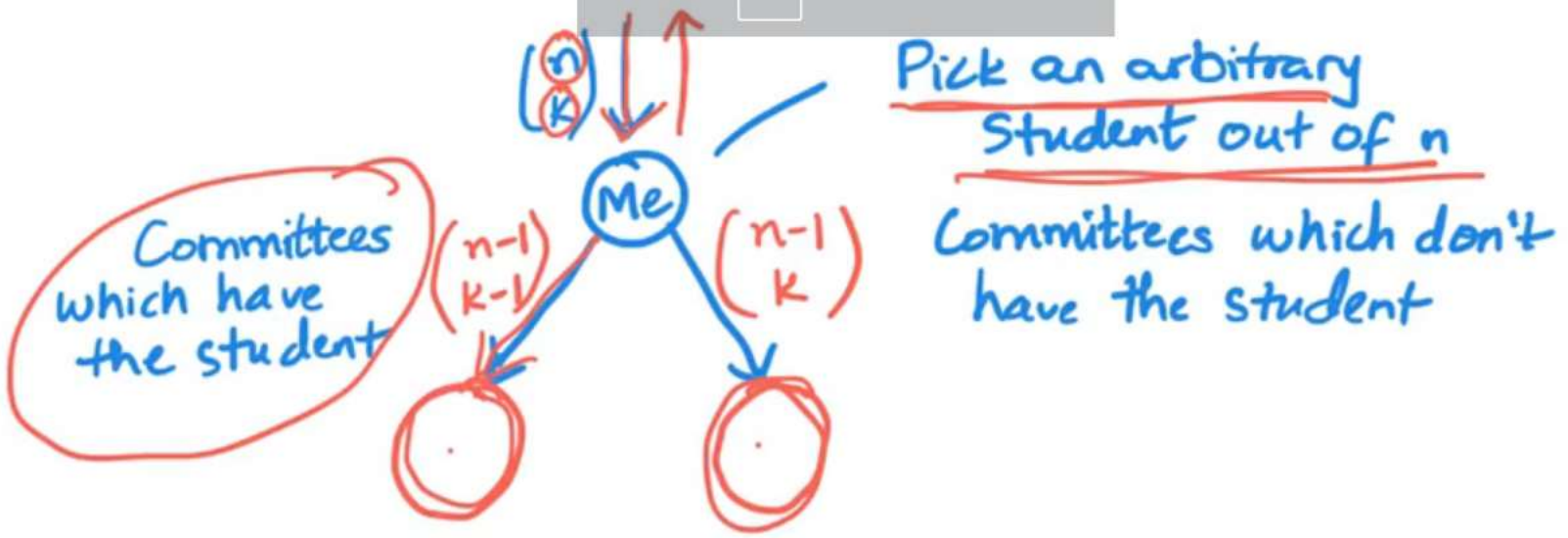
$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Number of ways to choose k objects out of n, where repetition is not allowed *and order is also not important*.

$$\binom{n}{n-k} = \frac{n!}{(n-k)!\,(n-\overline{n-k})!} = \frac{n!}{(n-k)!\,k!} = \binom{n}{k}$$

C(n,n) = ? 1 =
C(n,0) = ? 1

C(n,k) = C(n,n-k)

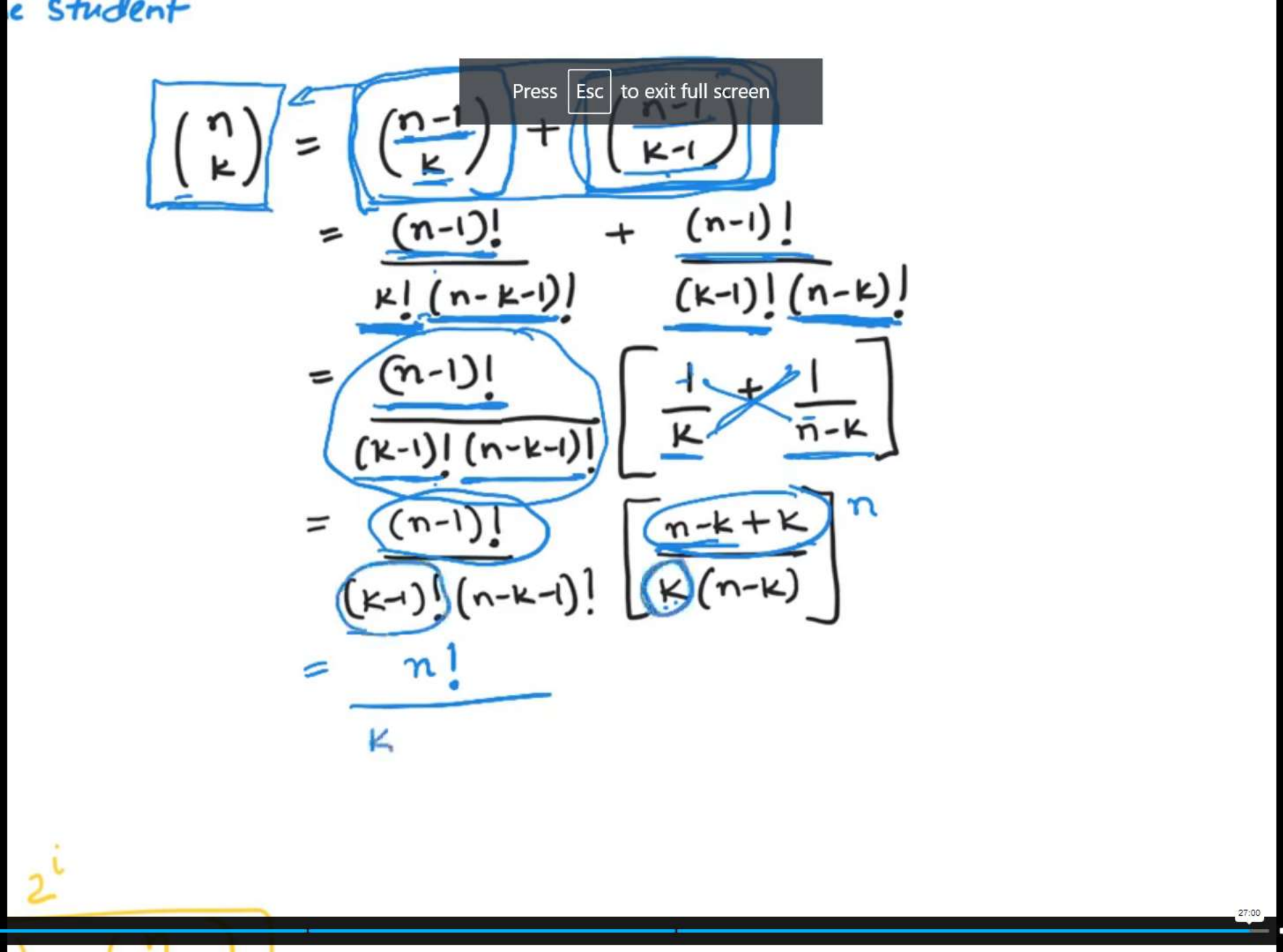


$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

Base Case: $k = 0$ or $k = n$

$$C(n,0) = C(n,n) = 1$$

e student



2. Recursive procedures

(While functions are mathematical entities, procedures are more algorithmic in character)

2. Recursive procedures

(While functions are mathematical entities, procedures are more algorithmic in character)

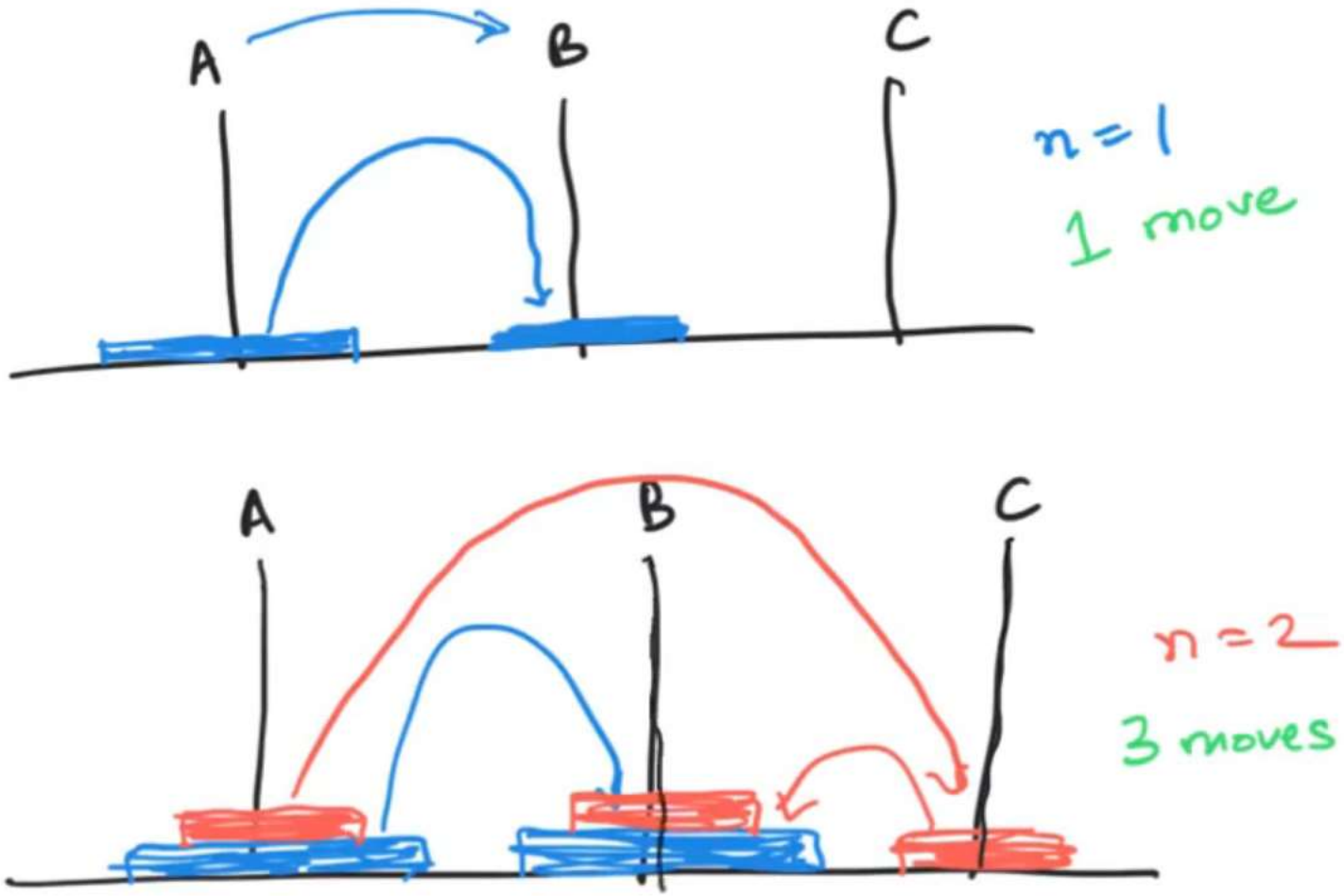
algorithmic in character)

Tower of Hanoi. Invented by French mathematician Edouard Lucas in the 1880s, the Tower of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in *La Nature* by the French mathematician Henri De Parville (as translated by the mathematical historian W. W. R. Ball):

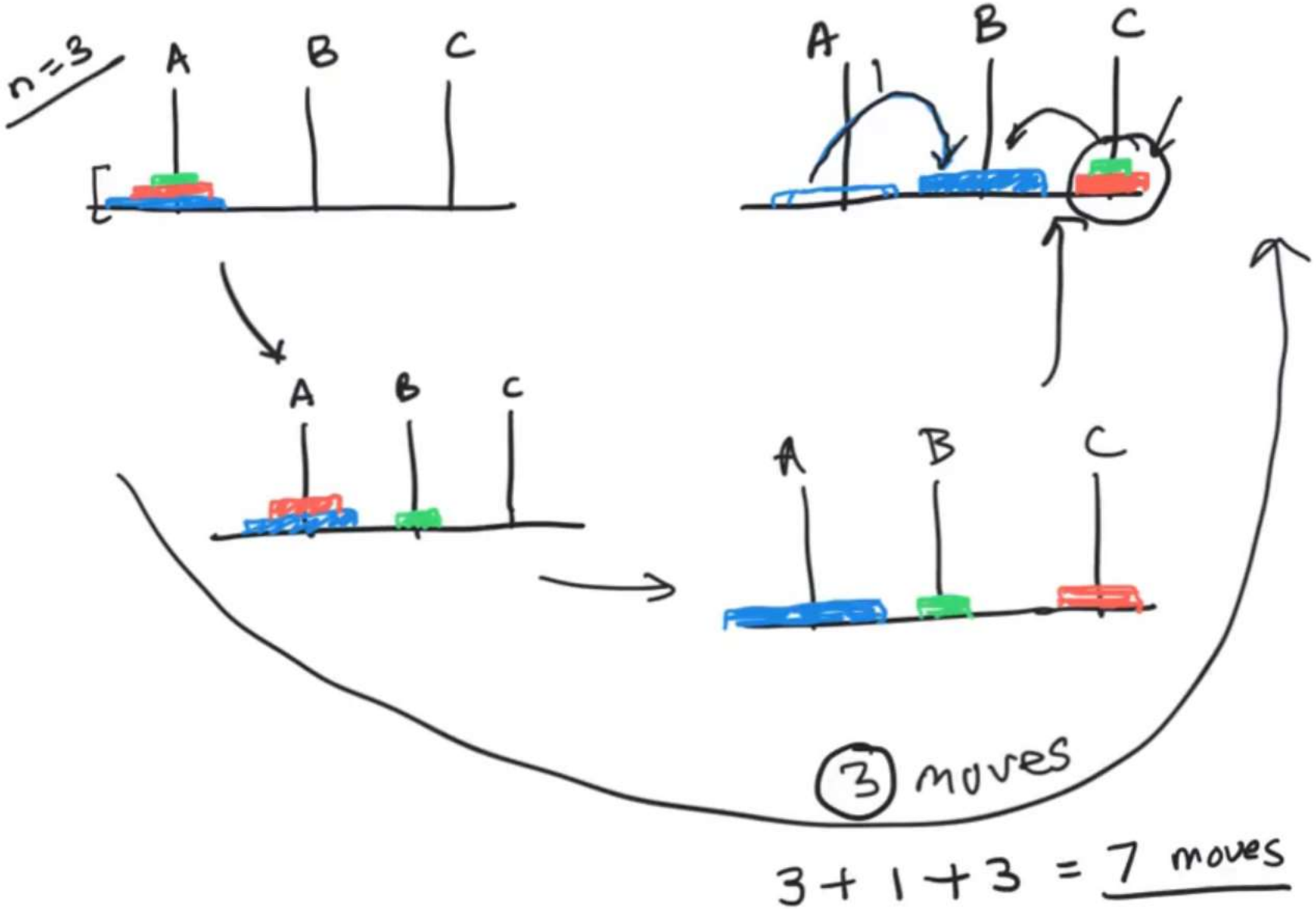
> In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Recursive Procedures
00:50

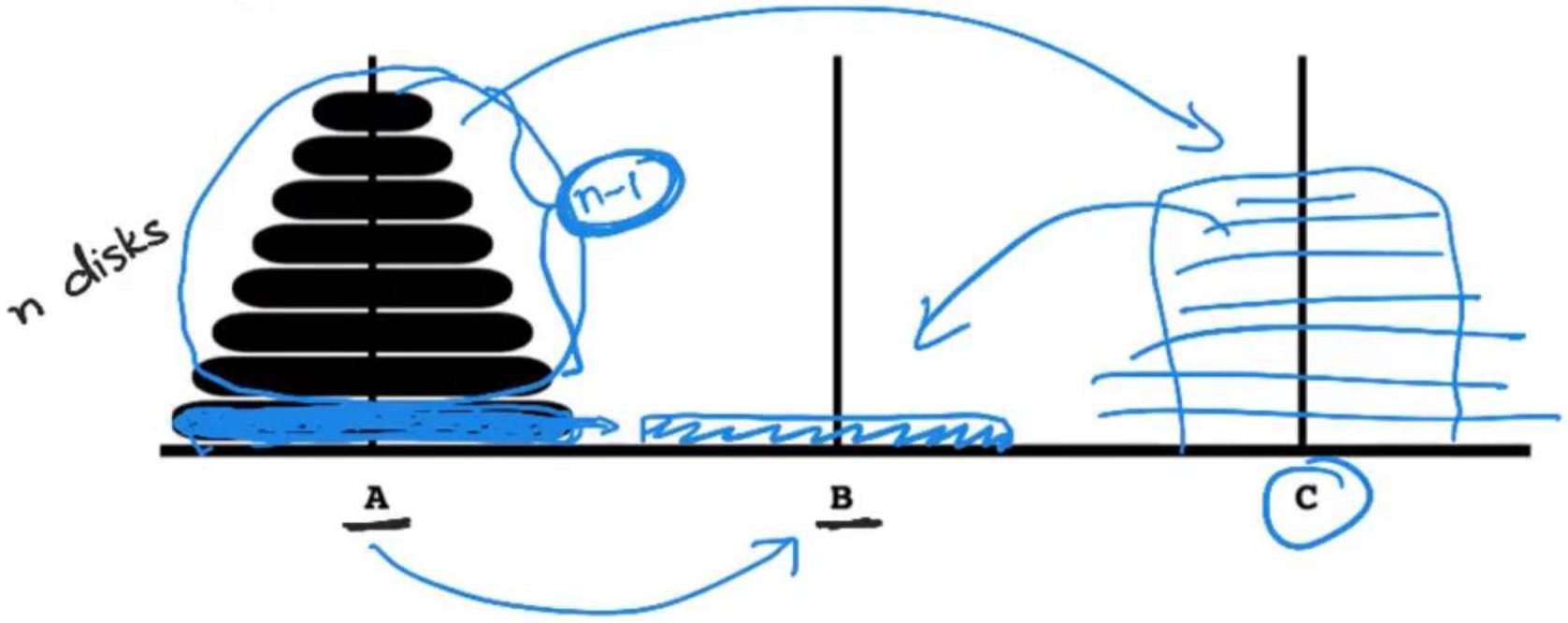
05:09

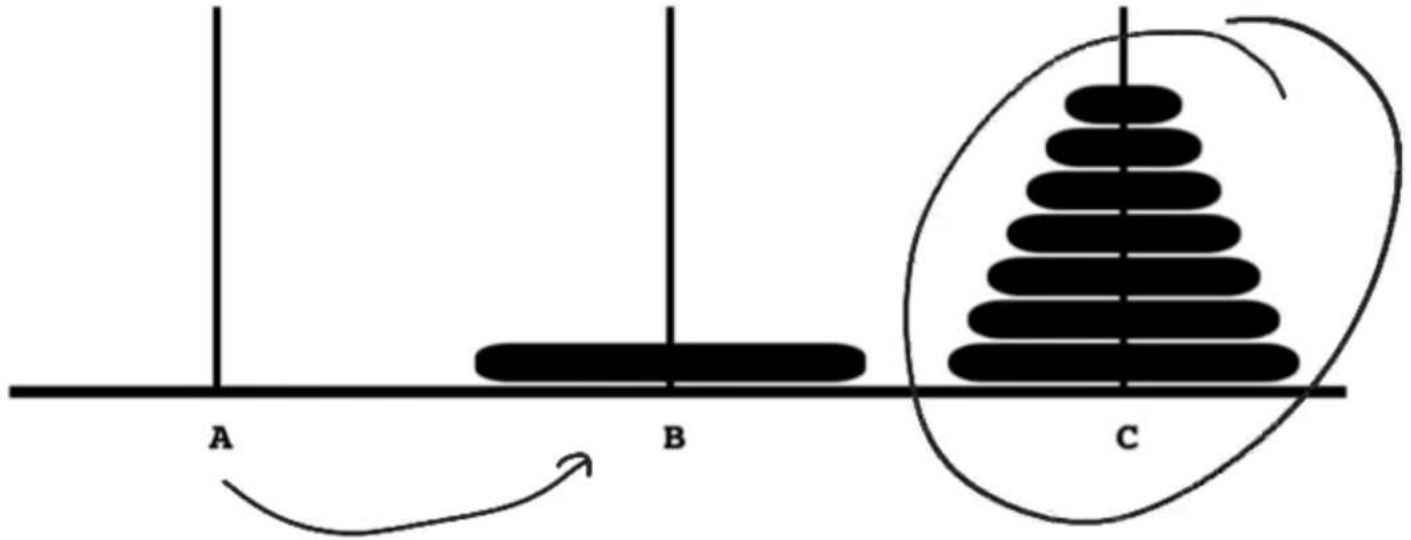


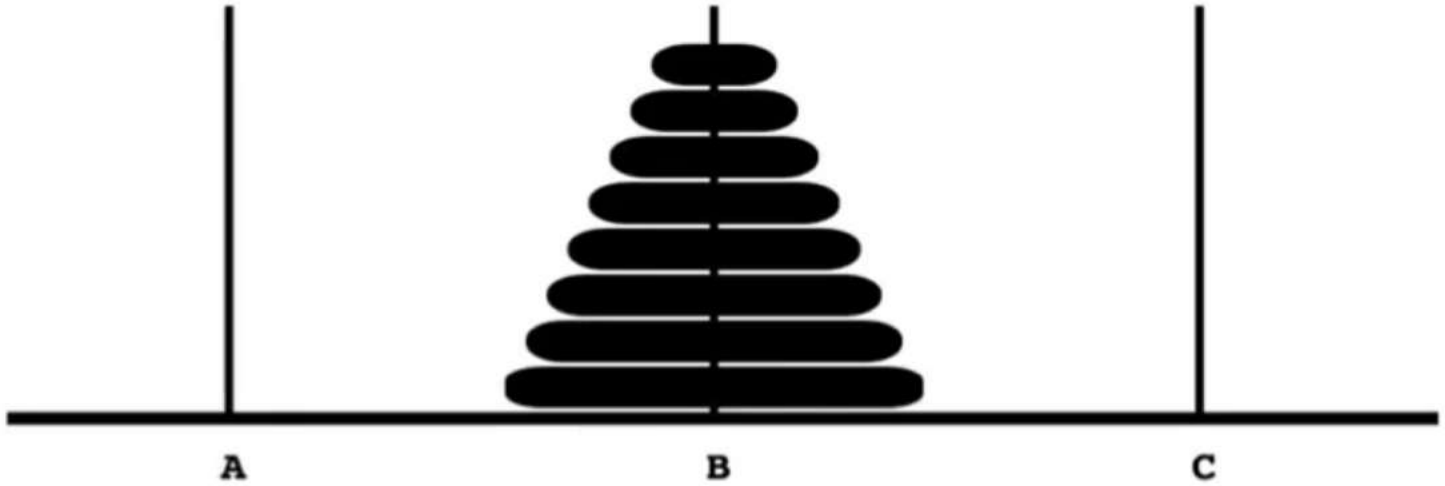
07:22


n=3
A
B
C
1
3 moves
3 + 1 + 3 = 7 moves

On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.






A
B
C

Strategy : Decrease and conquer

As a lazy manager, I will focus only on doing the (miniscule) work for the bottom disk.

My reports should do all the other work.

General strategy used by everyone: move n disks from src to dst (using aux)

if n = 1, move a single disk from src to dst.

else

move n-1 disks from src to aux

move a single disk from src to dst

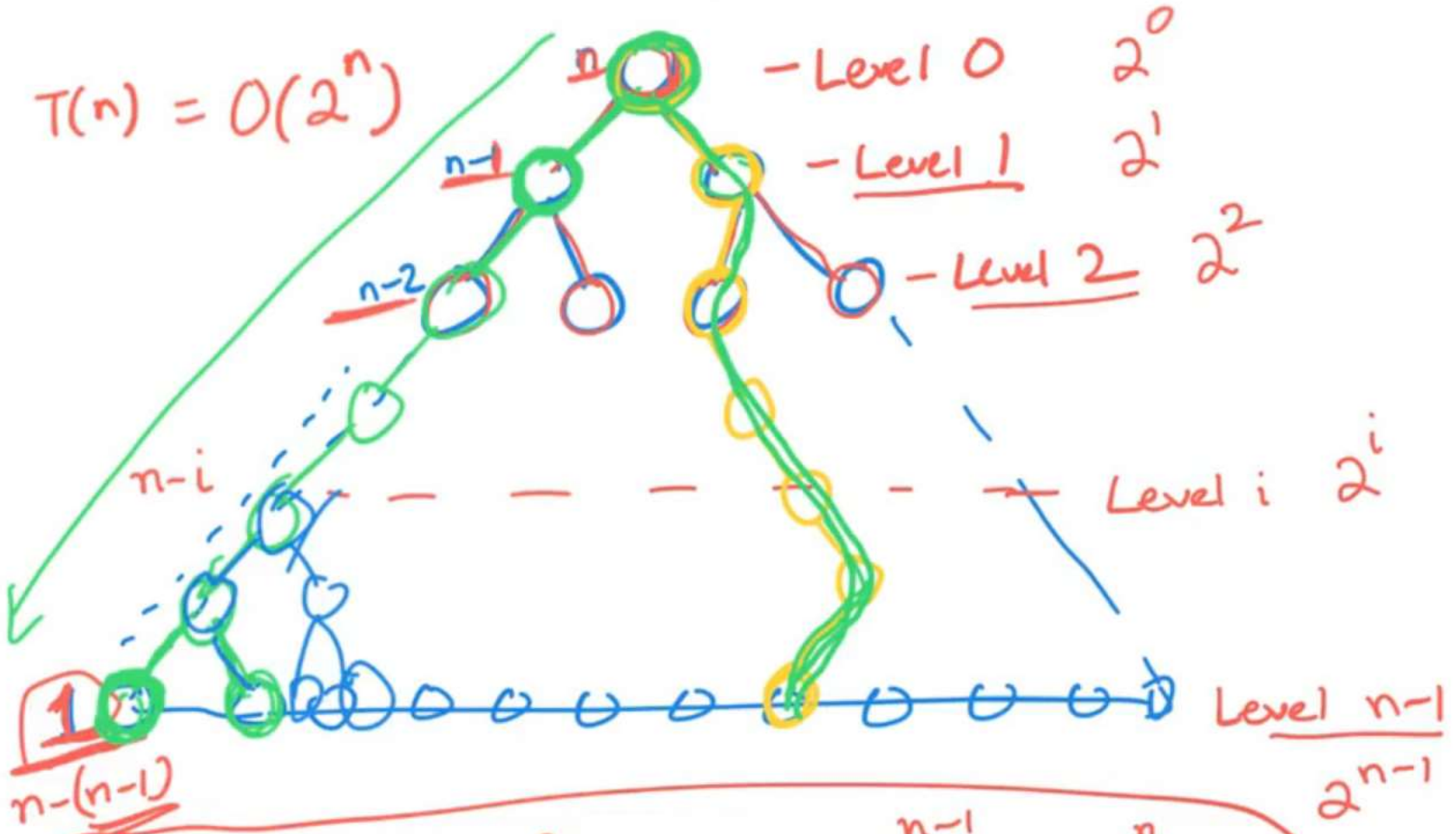
move n-1 disks from aux to dst.

```
def Tower of Hanoi (n, src, dst, aux):
    if n == 1:
        print "Move disk from" + src + " to " + dst
    else:
        Tower of Hanoi (n-1, src, aux, dst)
        print "Move disk from" + src + " to " + dst
        Tower of Hanoi (n-1, aux, dst, src)
```

Base: $T(1) = 1$

$T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1$$



$$T(n) = 2^0 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1$$

$$1 + 2 + 2^2 + \cdots 2^i = 2^{i+1} - 1$$

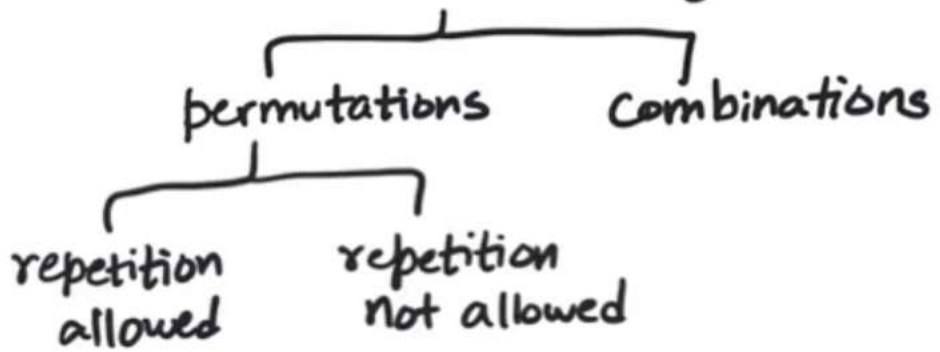
$T(1) = 1$

$T(2) = 2^2 - 1 = 3$

Exhaustive Search / Enumeration
Combinatorial

Given : A set of n distinct objects

enumerate all possible 'combinatorial objects'



0, 1

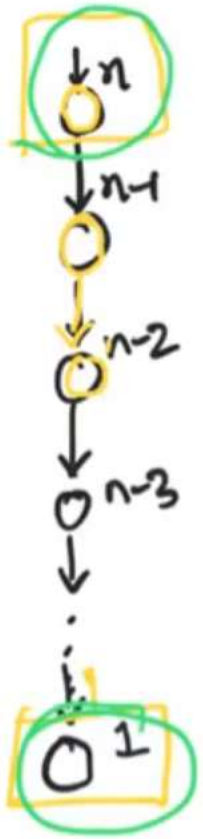
Enumerate all possible binary strings of length n:

2 x 2 x 2

1 or 1

```
for i in 0 to 1:
    for j in 0 to 1:
        for k in 0 to 1:
            print str(i) + str(j) + str(k)
```

000
001
010
011
100
101
110
111



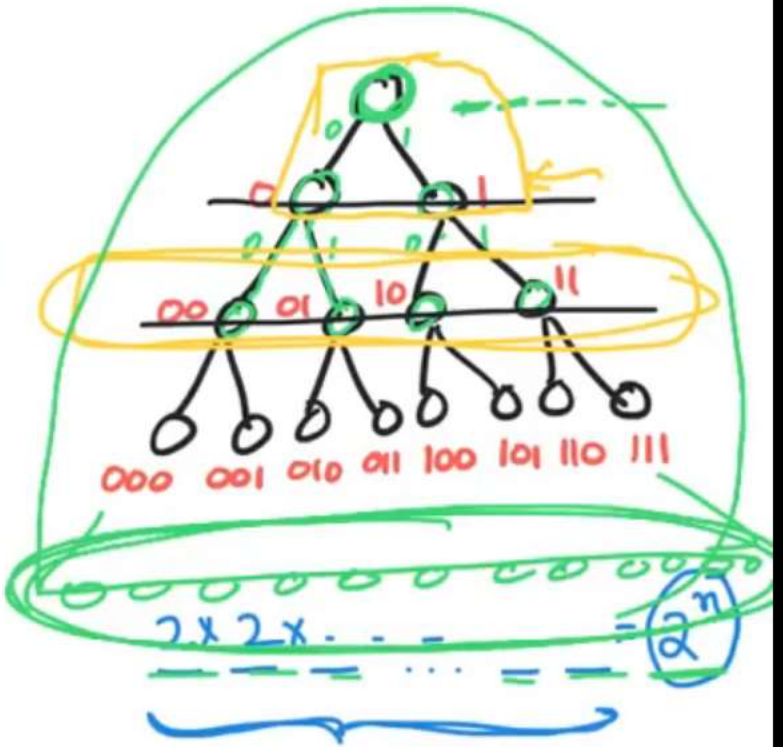

```
def binarystrings(n):
    if n==1:
        return ["0", "1"]
    else: # n>1
        prev = binarystrings(n-1)
        result = []
        for s in prev:
            result.append(s + "0")
            result.append(s + "1")
        return result
```

Space complexity $= O(2^n)$

$2^1 + 2^2 + 2^3 + \dots 2^n$

$= 2[1 + 2 + 2^2 + \dots 2^{n-1}]$

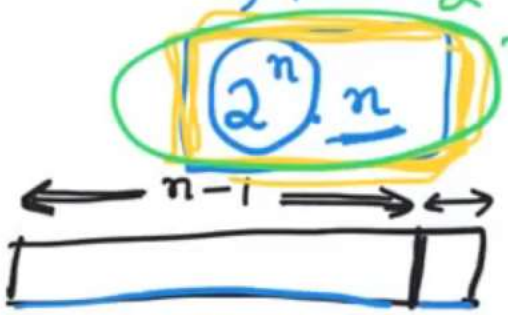
$= 2[2^n - 1] = 2^{n+1} - 2 = O(2^n)$



$2^0 + 2^1 + \dots 2^n$

$= 2^{n+1} - 1$

$\sim 2^{n+1}$



$\frac{2^n}{2^{n+1}} = \frac{1}{2}$

$= 2[2^n - 1] = 2^{n+1} - 2$

```
def binarystrings(n):
    result = ["0", "1"]
    for iter in 2 to n:
        newresult = []
        for each string s in result:
            newresult.append(s + "0")
            newresult.append(s + "1")
        result = newresult
    →
```

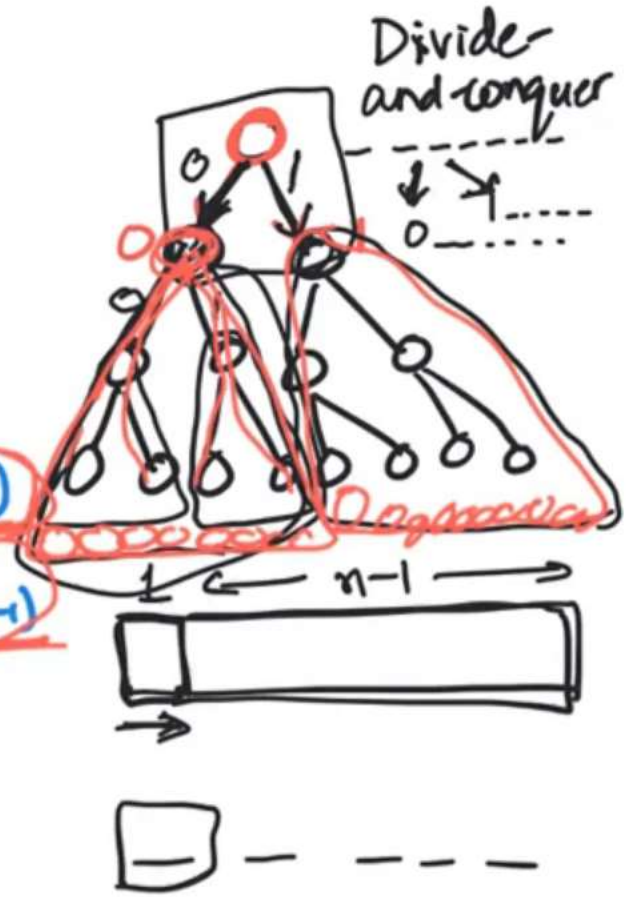
$n-1 \rightarrow n$

$1 \rightarrow 2$

$2 \rightarrow 3$

$n-1 \rightarrow n$

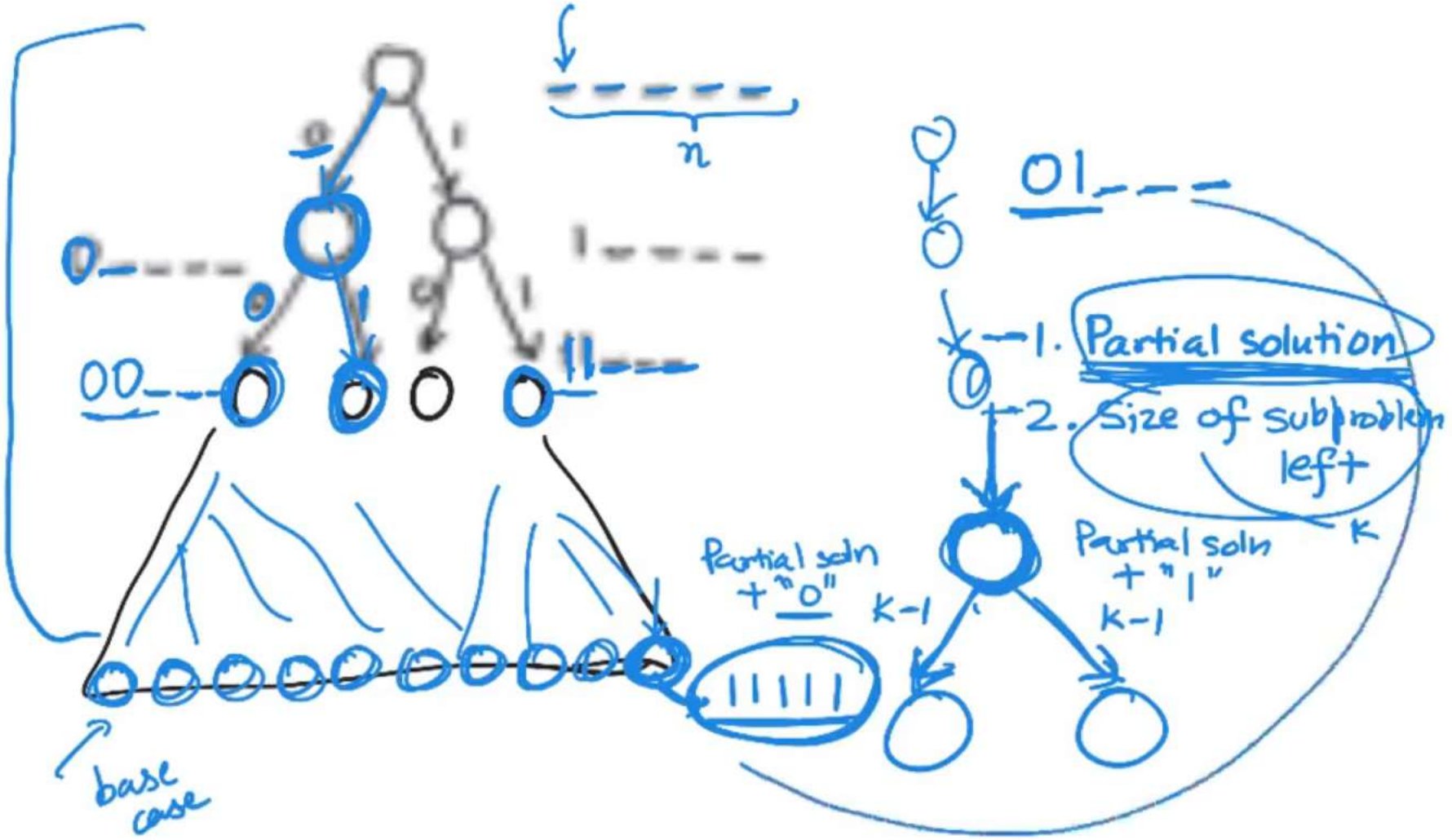
```
def binarystrings(n):
    if n == 1:
        print ["0", "1"]
    else:
        print "0" + binarystrings(n-1)
        print "1" + binarystrings(n-1)
```



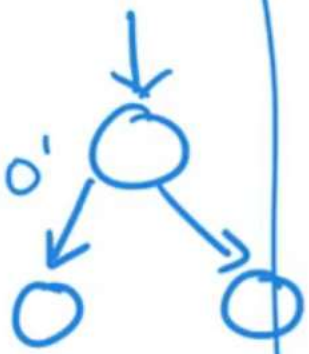
n = 5

0 [ ]

1 [ ]

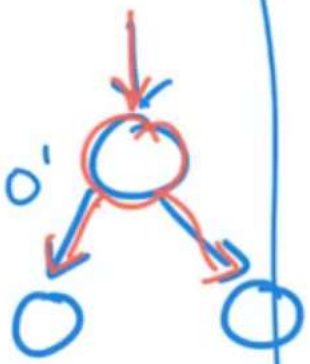

n
0
1
00
01
11
base case
Partial soln + "0"
k-1
Partial soln + "1"
k-1
01---
1. Partial solution
2. Size of subproblem left
k

```
def binarystrings(slate, n):
    Base case
    if n==0:
        print slate
    else:
        Recursive case
        binarystrings(slate+"0", n-1)
        binarystrings(slate+"1", n-1)
```



binarystrings("", 100)

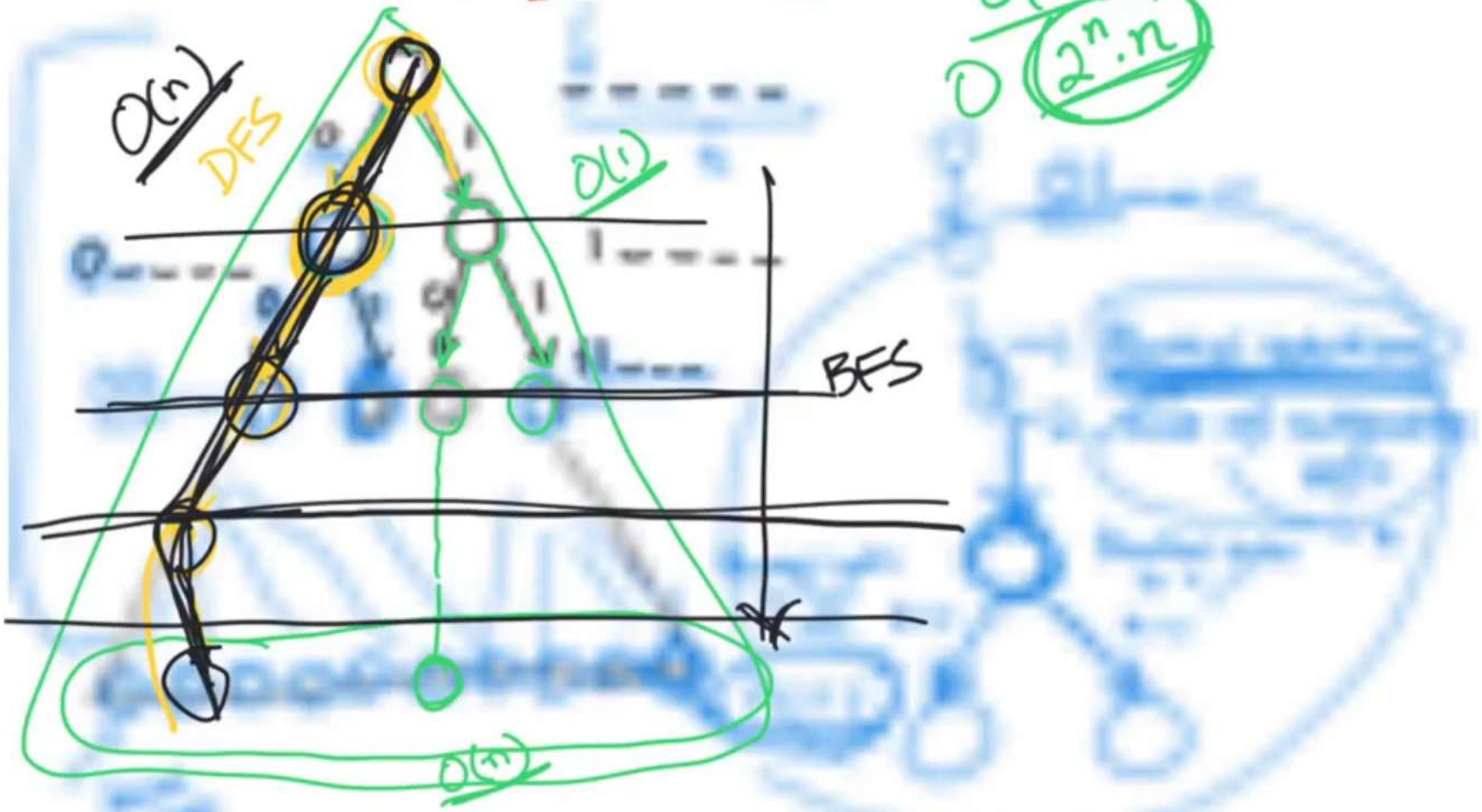
```
def bs_helper(slate, n):
    Base case
    if n == 0:
        print slate
    else:
        Recursive case
        bs_helper(slate + "0", n-1)
        bs_helper(slate + "1", n-1)

def binary_strings(n):
    bs_helper("", n)
```



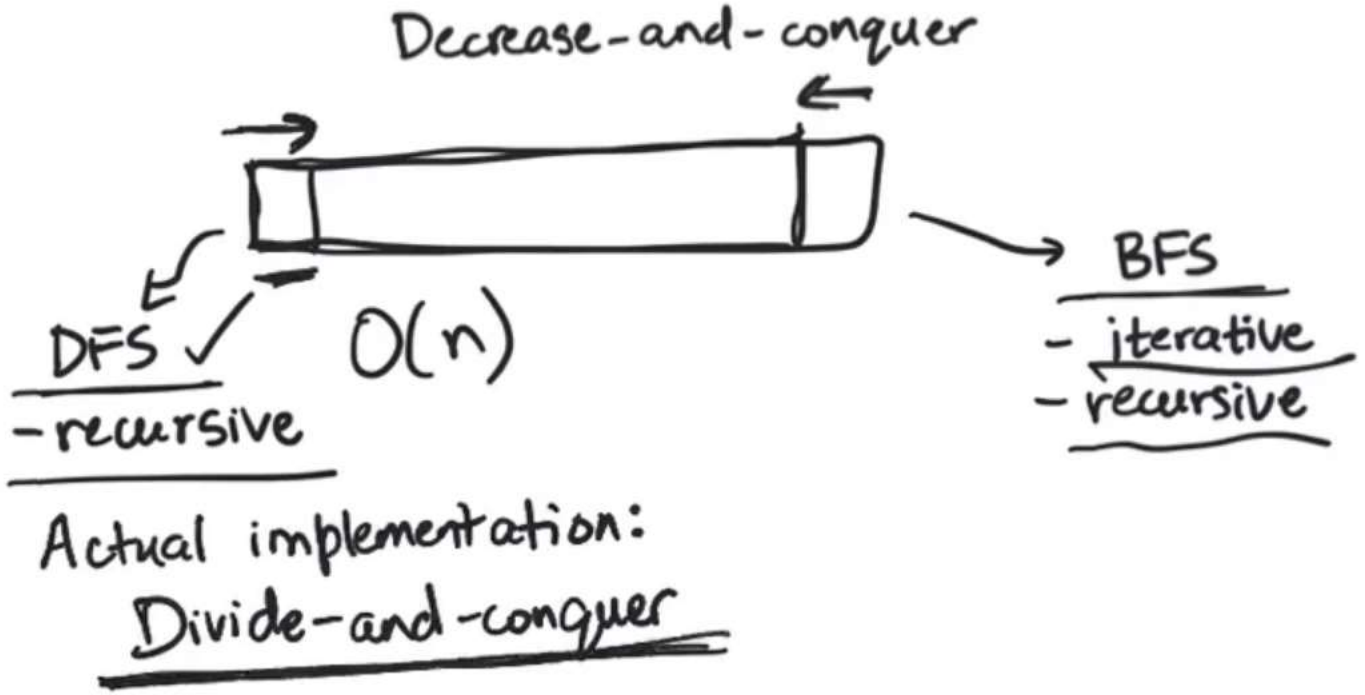
n = 3

0 [ ]

1 [ ]



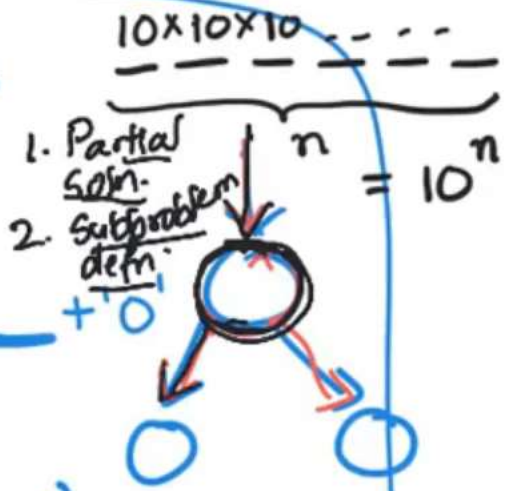

```
def bs_helper (slate, n):
```


Decrease-and-conquer
DFS ✓
O(n)
- recursive
BFS
- iterative
- recursive
Actual implementation:
Divide-and-conquer

Print all ~~binary~~ decimal strings of length n.

0...9

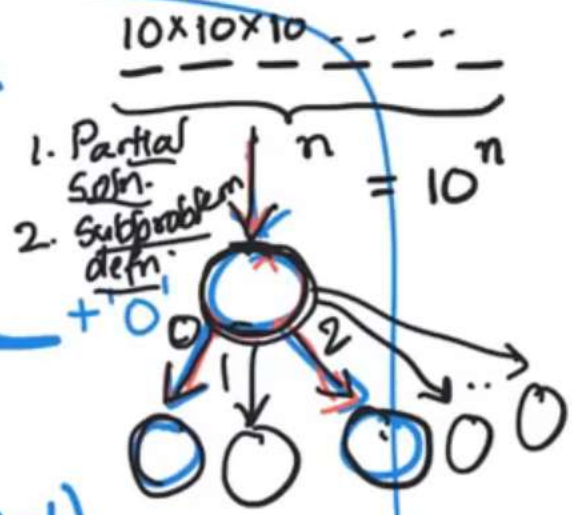
```
def bs_helper(slate, n):
    Base case:
    if n == 0:
        print slate
    else:
        Recursive case:
        bs_helper(slate + "0", n-1)
        bs_helper(slate + "1", n-1)

def binary_strings(n):
    bs_helper("", n)
```



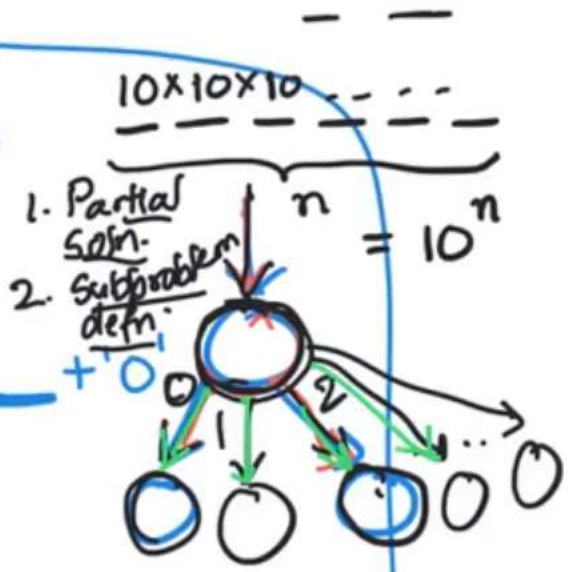
Print all ~~binary~~ decimal strings of length n.

0...9

```
def ds_helper (slate, n):
    Base case [ if n==0:
                    print slate
    else:
    Recursive case [ ds_helper (slate+"0", n-1)
                     ds_helper (slate+"1", n-1)
                     .
                     .
                     ds_helper (slate+"9", n-1)

def binary_strings (n):
    bs_helper ("", n)
```



```
def ds_helper(slate, n):
    Base case
    if n == 0:
        print slate
    else:
        Recursive case
        for i in 0 to 9:
            dshelper(slate + str(i), n-1)
```




```
def decimal strings(n):
    bshelper("", n)
```

00
01
02
⋮
99

Permutations with repetitions : Binary strings
Decimal strings

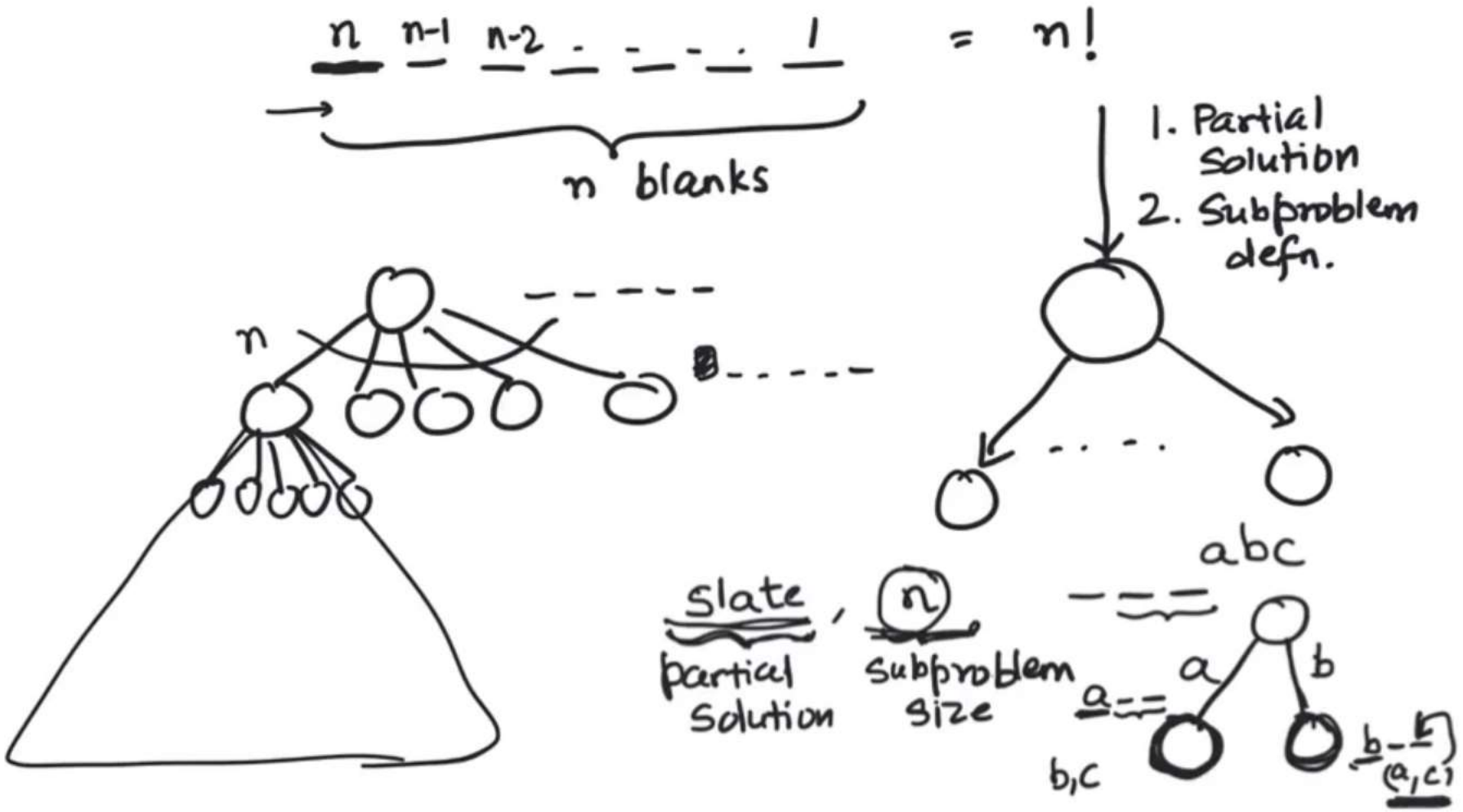
Permutations without repetitions :

Given a set (or string) with n distinct characters,
print all permutations (of size n,
no repetitions
allowed)

e.g. Input : abc n

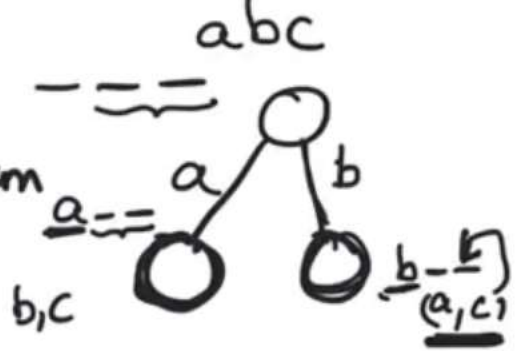
Output :
abc
acb
bac
bca
cab
cba

n!


n n-1 n-2 _ _ _ _ _ 1
= n!
n blanks
1. Partial solution
2. Subproblem defn.
n
abc
slate, (n)
partial solution
subproblem size
a
b
b,c
(a,c)

slate, n

partial solution — subproblem size




```
def phelper(slate, array):
    if length(array) == 0:
        print slate
    else:
        for i in 0 to length(array)-1:
            phelper(slate + array[i], array[:i] + array[i+1:])
```

Base case

first i-1 char

Subproblem definition

starting from position i+1

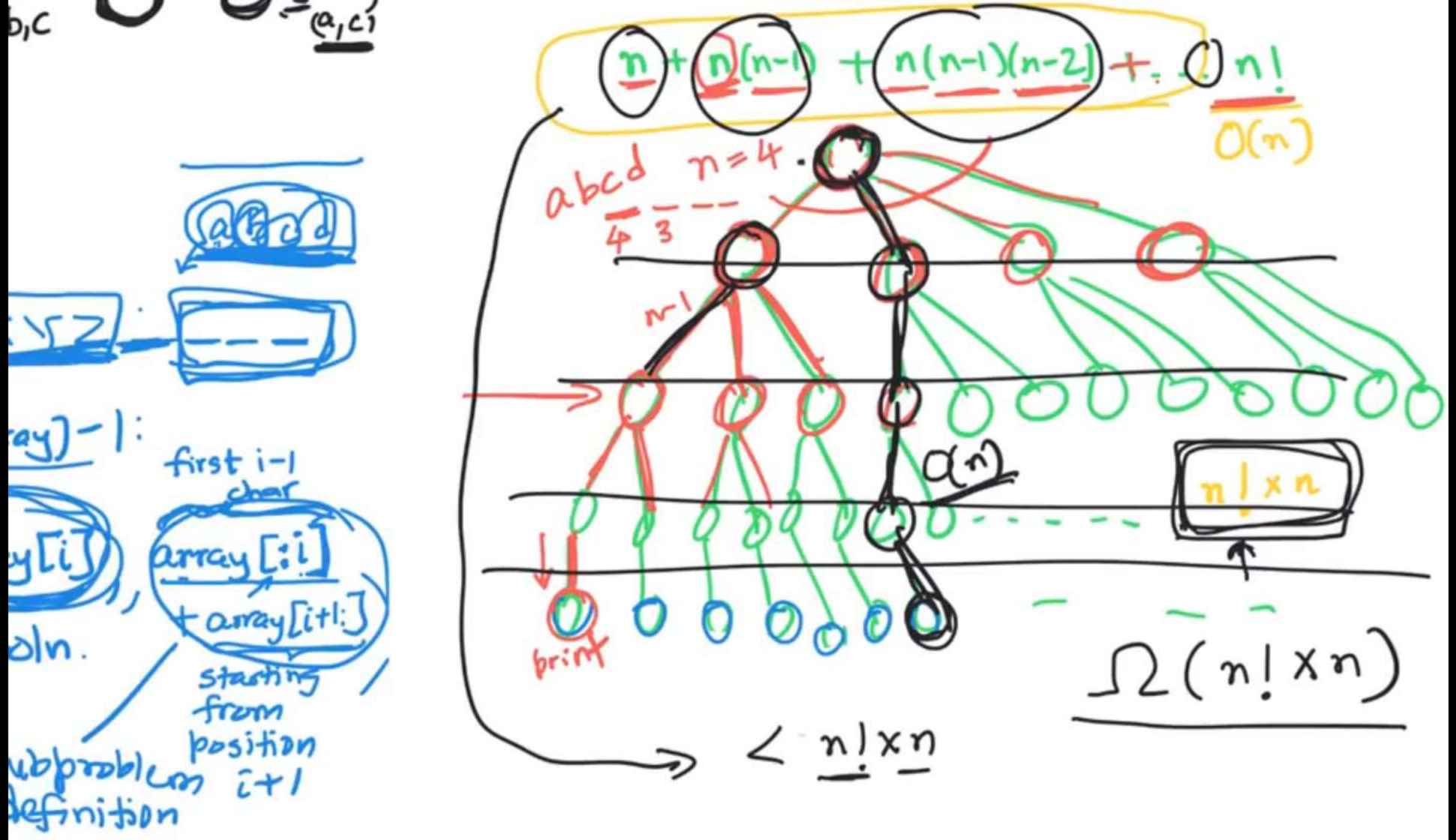
```
def phelper(slate, array):
    if length(array) == 0:
        print slate
    else:
        for i in 0 to length(array)-1:
            phelper(slate + array[i], array[:i] + array[i+1:])

def printpermutations(s):
    phelper("", s)
```

Base Case

slate + array[i] : partial soln.

array[:i] + array[i+1:] : first i-1 char, starting from position i+1

Subproblem definition


n + n(n-1) + n(n-1)(n-2) + ... n!
O(n)
abcd n=4
4 3
n-1
O(n)
n! x n
print
Ω(n! x n)
< n! x n
first i-1 char
array[:i]
+ array[i+1:]
starting from position i+1
subproblem definition

From permutations to combinations:

Given: A set S of n distinct numbers,
print (enumerate) all its subsets.

e.g., if S = {1, 2, 3}

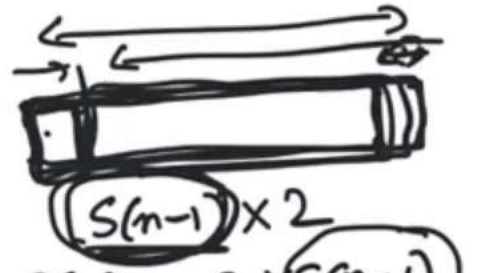
Output: {}, {1}, {2}, {3},
{1,2}, {2,3}, {3,1},
{1,2,3}

{1,3}

2^{n-1}

② $2 \cdot 2 \cdots 2 = 2^n$

$2 \times 2 \times \cdots \times 2$

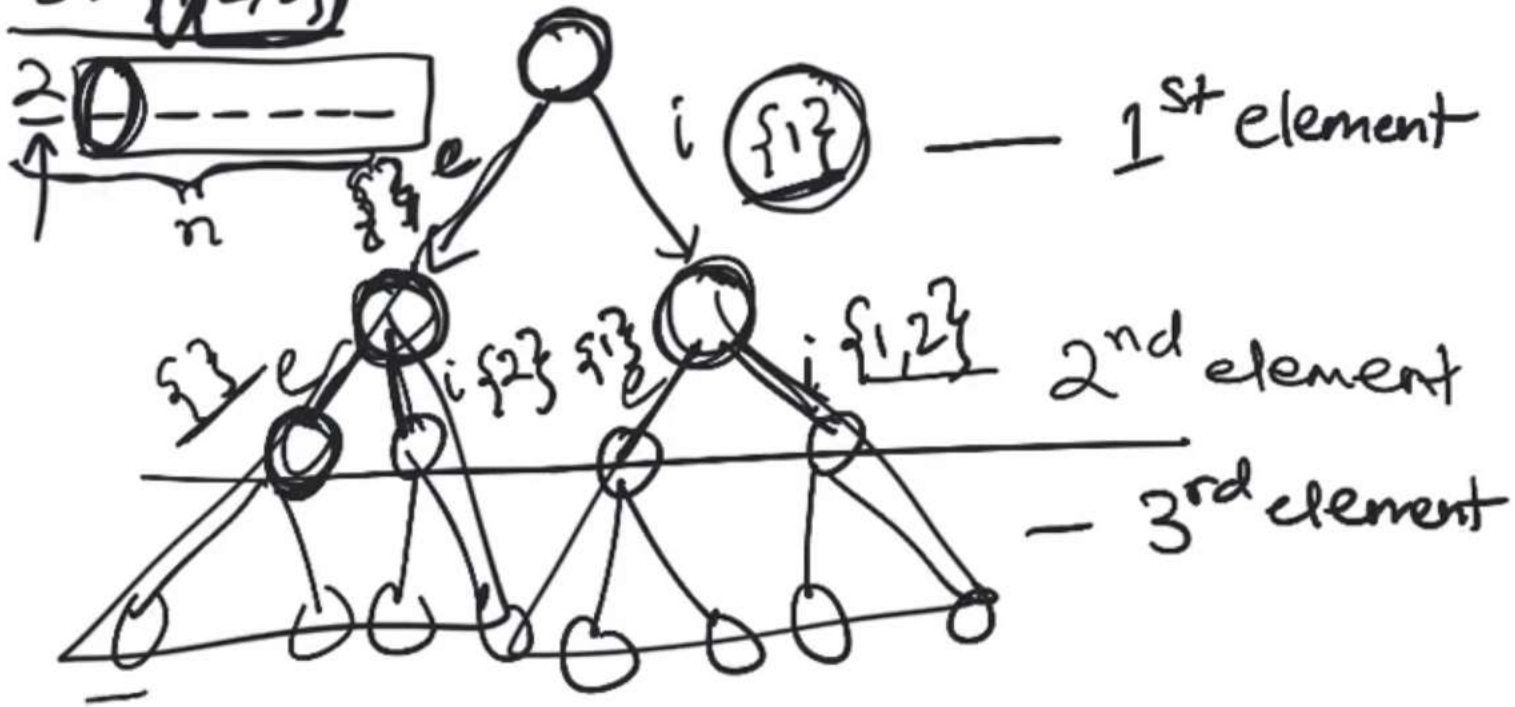


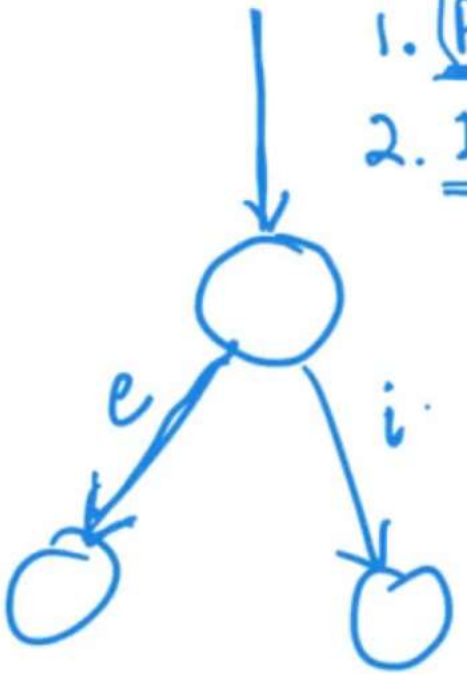
$S(n) = 2 \times S(n-1)$
$= 2 \times 2 \times \cdots = 2^n$

$S(n) = 2 \times S(n-1)$
$= 2 \times 2 \times \cdots = 2^n$

Inclusion exclusion choice pattern for subsets/combinations:

$S = \{1,2,3\}$





1. Partial solution
2. Definition of subproblem

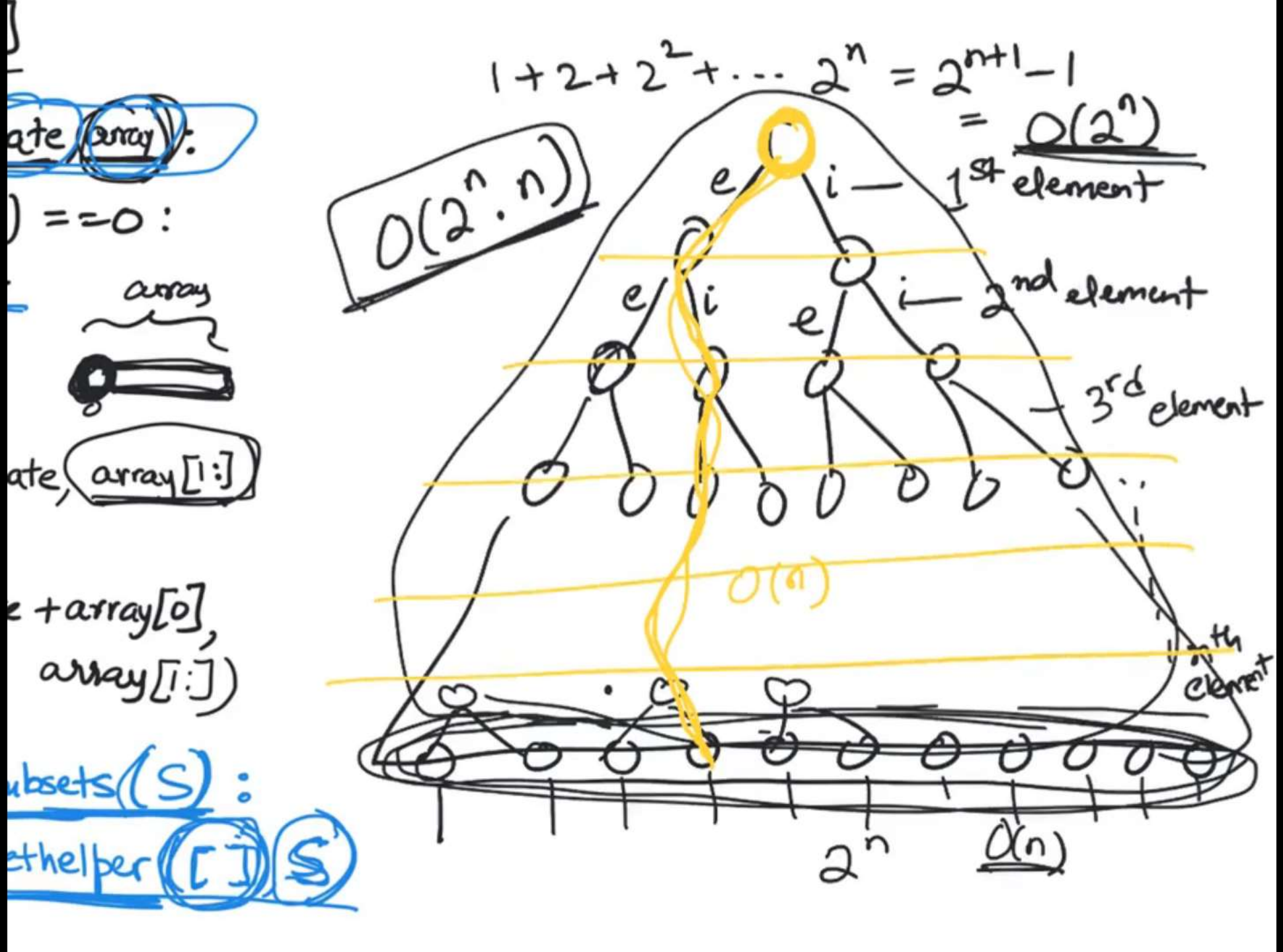
bag of elements
I have to look at to generate all subsets.

ution

of subproblem

g of elements
have to look
to generate
subsets -


```
def subsethelper(slate, array):
    Base case:
        if length(array) == 0:
            print slate
        else:
    Recursive case:
        // exclude
        subsethelper(slate, array[1:])
        // include
        subsethelper(slate + array[0], array[1:])

def printsubsets(S):
    subsethelper([], S)
```


ate (array):
) ==0:
array
ate, array[1:]
e +array[0],
array[1:])
ubsets(S):
ethelper([],S)
$1+2+2^2+\cdots 2^n = 2^{n+1}-1$
$= O(2^n)$
$O(2^n \cdot n)$
1st element
2nd element
3rd element
nth element
O(n)
2^n
O(n)

S = [, ... ,]

```
def subsethelper(slate, array):
    if length(array) == 0:
        print slate
    else:
        // exclude
        subsethelper(slate, array[1:])
        // include
        subsethelper(slate + array[0], array[1:])

def printsubsets(S):
    subsethelper([], S)
```

array

$1 + 2 + 2^2 + \ldots$

$O(2^n \cdot n)$