Algorithm and Object-Oriented Programming for Modeling
Part 1: Introduction to Data Structure and Algorithm

MSDM 5051, Yi Wang (王一), HKUST

An *Elective* Course of the MSDM Program

Not prerequisite of other courses

If you have already learned {data structure, algorithm, OO}

then you don't need to learn it again (review?)

A lower bound elective

If it's too simple, spend more time in other things ^_^

But is important:

- Ideas of how to solve classes of common problems
- Use standard method eases software engineering
- Algorithm for interviews

You should know one programming language in advance.

Preferably Python. If not, learn a language is simple.

If you still struggle with languages:

Can take the course now,

but need more efforts.

三百六十行，行行出 **bug**

Standalone

Cluster

Hot swap

RAID 0

RAID 1

RAID 5

RAID 0+1

Consider more practice and projects.

Plan of Lectures:

Data structure

Sorting algorithms

BFS, DFS and shortest path

Dynamic programming

Object Oriented (OO) programming

Design Patterns

Mock Interview (MI)

Grading:

Attendance 5%

Quiz 1: 15%

Quiz 2: 15%

Quiz 3: 15%

Interviewer: 3%

Interviewee: 7%

Project 1: 20%

\+ Project 2: 20%

100%

Instruction language: Python 3

Instruction environment:

VS Code + Anaconda on Windows

(You may use anything else if you prefer)

References:

- Grokking Algorithms, Bhargava
- Data Structures and Algorithms Using Python, Necaise
- MIT Open Course, 6.006, Introduction to Algorithms ✱
- (In Chinese, in C++) 邓俊辉， MOOC 上， MOOC 下
- Introduction to Algorithms (CLRS), Cormen, Leiserson, Rivest, Stein (best in the field, but not simple)
- Python 3 Object-oriented Programming, Phillips ✱
- Design Patterns (GoF), Gamma, Helm, Johnson, Vlissides
- TheAlgorithms/Python @ Github

Part I: Data Structure & Introduction to Algorithms

I-a: Data structure

We understand data structures
ever since we were kids

Linear data structures:

Random access vs sequential access

1	January
2	February
…	…
12	December

Array

Linear data structures:

Random access vs sequential access

1	January
2	February
…	…
12	December

Array

vs. linked list

Linear data structures:

Random access vs sequential access

1	January
2	February
…	…
12	December

Array

vs. linked list

先帝創業未半 → 而中道崩殂 → 今天下三分 → 益州疲弊 → …

Linear data structures:

Random access vs sequential access

1	January
2	February
…	…
12	December

Array

vs. linked list

release from memory

先帝創業未半 → 而中道崩殂 → 今天下三分 → 益州疲弊 → …

↓

越明年 → 政通人和 → 百廢具興 → …

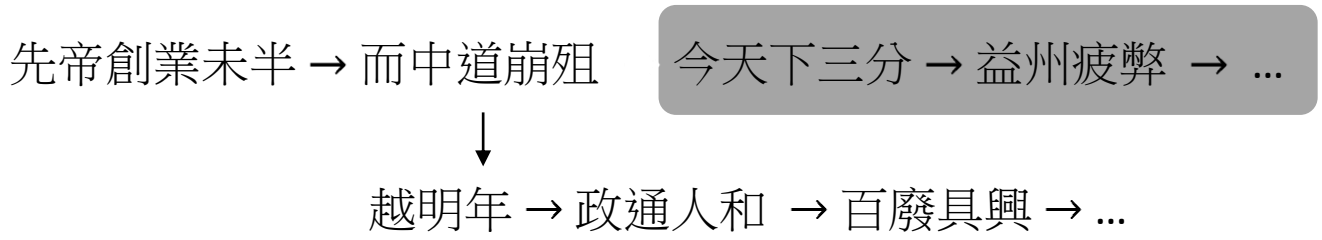
Inverse question: How to prevent such modification?

Each block contains some info about the previous block

-> Cannot modify a few elements

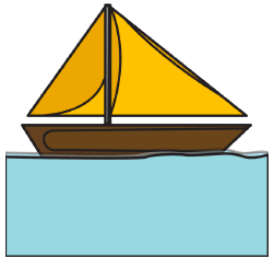
without changing the whole chain

-> blockchain



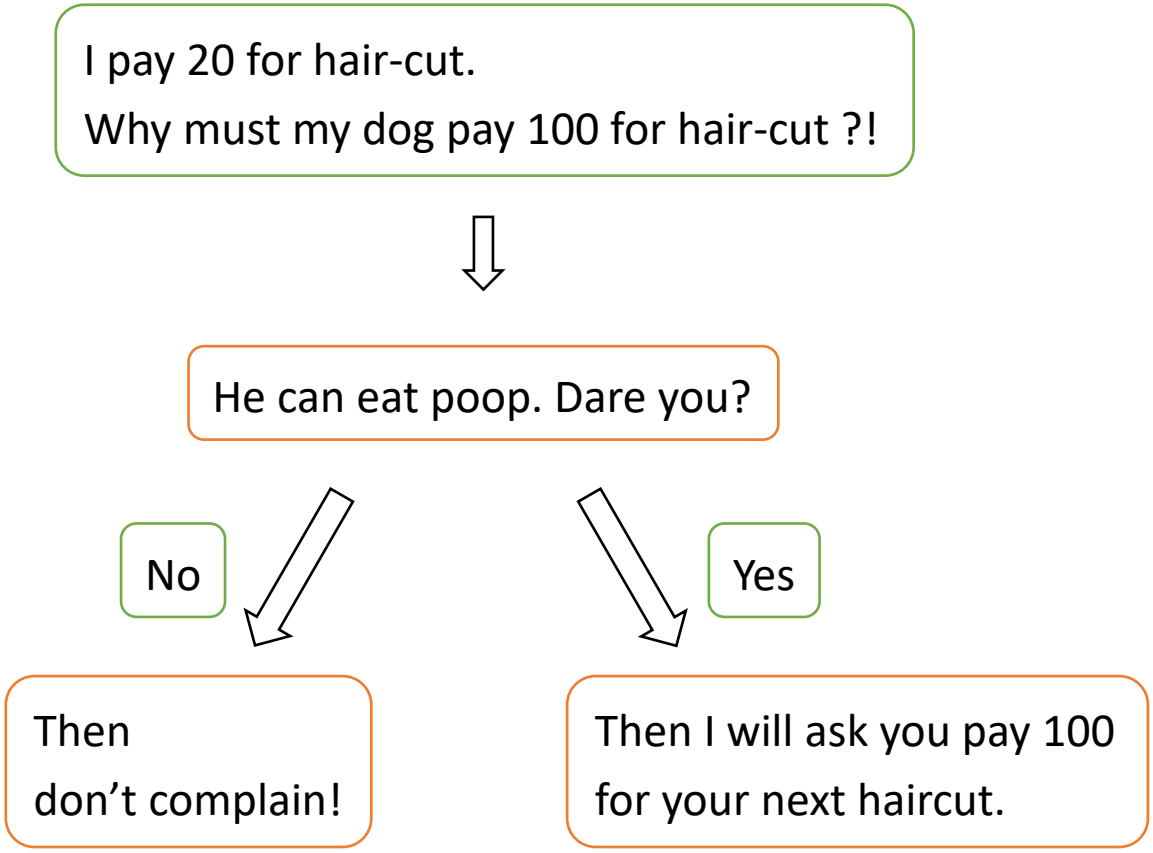
Number Memory Peg System

How to remember any 10 items in order.
See https://youtu.be/6i2xqWFIFz8 for instructions.

1	2	3	4	5
 paint brush	 swan	 heart	 yacht	 hook
6	**7**	**8**	**9**	**10**
 bomb	 cliff	 hourglass	 balloon	 knife + plate

Non-linear data structure: trees

Non-linear data structure: trees



Graphs: remembering a map

Data structure in computers:

Similar, just more stupid.

List (array) & Linked List

List

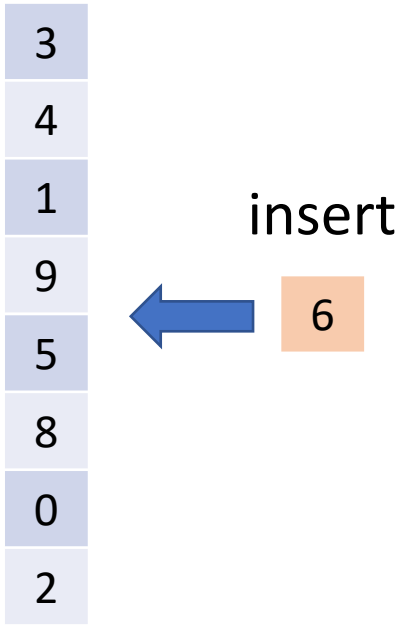
Memory address (simplified)

1	→	3
2	→	4
3	→	1
4	→	9
5	→	5
6	→	8
7	→	0
8	→	2

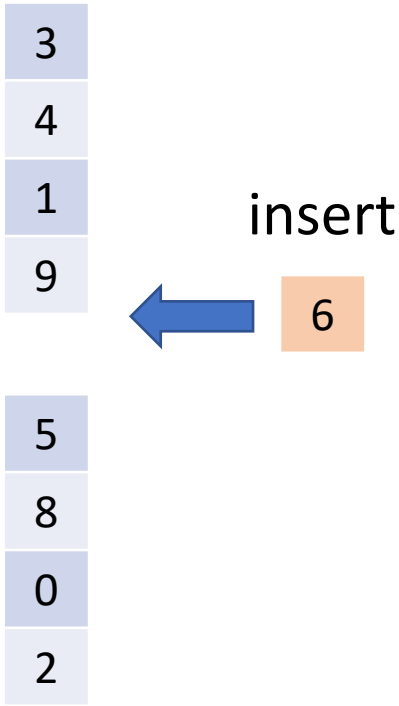
address data

Operation	Example	Time
Construct	A = [3, 4, …]	O(n)
Access	A[5]	O(1)
Insert	A.insert(5, 6)	?

Do we need data structure other than lists?

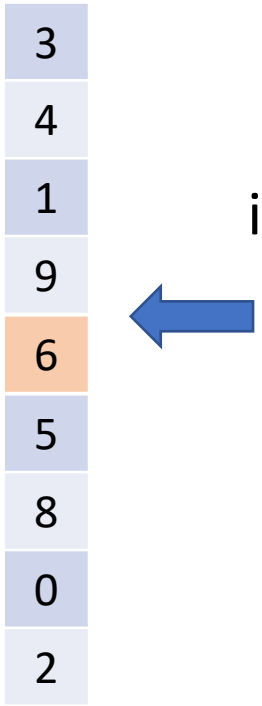


Do we need data structure other than lists?



1. Move elements down: O(n)

Do we need data structure other than lists?



1. Move elements down: O(n)
2. Actual insertion: O(1)

Thus, total: O(n) for insertion

Similar for deletion

So, if we need

Frequent insertion/deletion

Not frequent random access

We shouldn’t use arrays.

List

Memory address (simplified)

address		data
1	→	3
2	→	4
3	→	1
4	→	9
5	→	5
6	→	8
7	→	0
8	→	2

Operation	Example	Time
Construct	A = [3, 4, …]	O(n)
Access	A[5]	O(1)
Insert	A.insert(5, 6)	O(n)

List

Memory address (simplified)

address		data
1	→	3
2	→	4
3	→	1
4	→	9
5	→	5
6	→	8
7	→	0
8	→	2

Operation	Example	Time
Construct	A = [3, 4, …]	O(n)
Access	A[5]	O(1)
Insert	A.insert(5, 6)	O(n)
Delete	A.delete(5)	O(n)
Search	if 9 in A	O(n)
Append	A.append(7)	?
Pop	A.pop()	?

Initially Table is Empty and size of array is 0

Insert Item1

Table doubling

Insert Item2

Insert Item3

Insert Item4

Insert Item5

Insert Item6

Insert Item7

Next over flow will happen at the time of inserting 9,when table size would become 16

List

Memory address (simplified)

address		data
1	→	3
2	→	4
3	→	1
4	→	9
5	→	5
6	→	8
7	→	0
8	→	2

Operation	Example	Time
Construct	A = [3, 4, …]	O(n)
Access	A[5]	O(1)
Insert	A.insert(5, 6)	O(n)
Delete	A.delete(5)	O(n)
Search	if 9 in A	O(n)
Append	A.append(7)	?
Pop	A.pop()	?

our default

Worst: O(n)
Amortized worst: O(1)
(Table expansion tech)

Remark: Linear lists may be realized in different ways

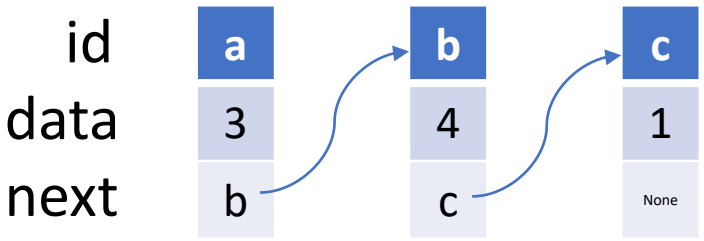
For example, lists and tuples (see also set {…} later) in python:

List	Tuple
A = [1, 2, 3]	A = (1, 2, 3)
Mutable	Immutable
A[1]=0	(N/A)
Built-in append, insert, pop, remove, reverse, sort	Those methods not built-in.
Append O(1) amortized	Append O(n)
Harder to debug B=A A[1]=5 (now B[1] also changed)	Easier to debug

(see also array.array, np.array)

Other examples: in C++, c-style array vs std array

Linked list

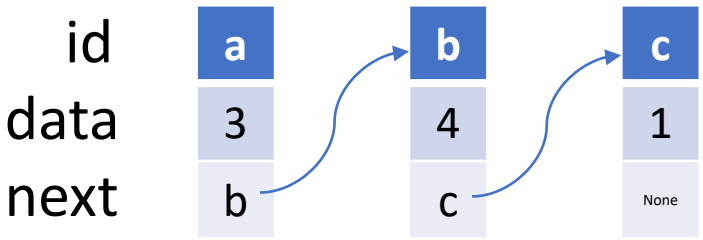



```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

Note: id(obj) in Python is similar (in limited sense) to pointers in C

```
struct LinkedList{
    int data;
    struct LinkedList *next;
};
```

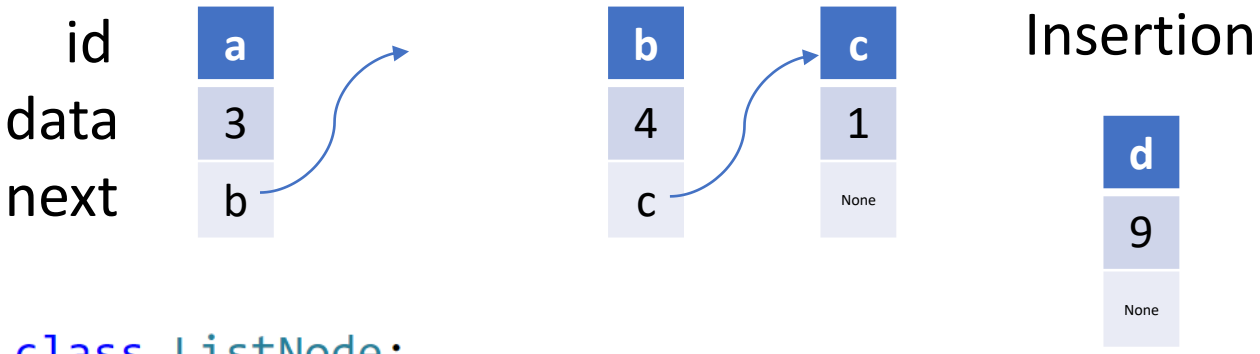
Linked list



Insertion

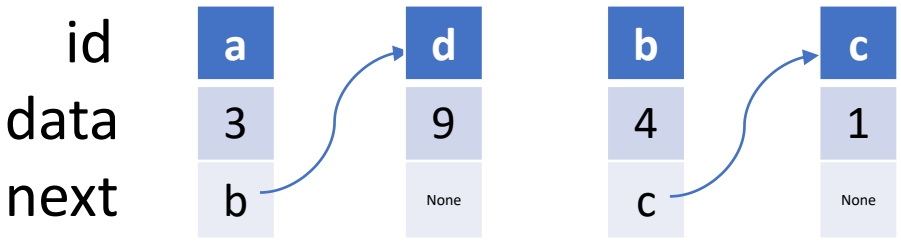

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

Linked list




```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

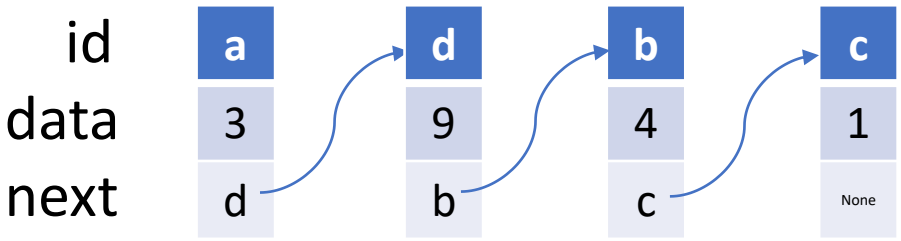
Linked list



Insertion

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

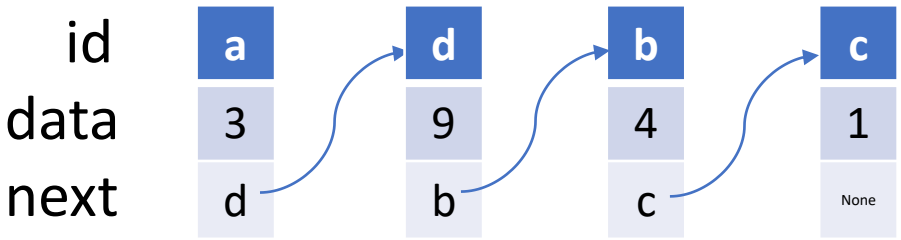
Linked list



Insertion: O(1)

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

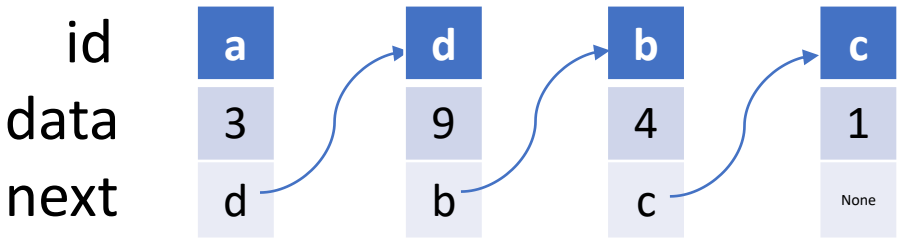
Linked list




```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

sometimes "head"

Linked list



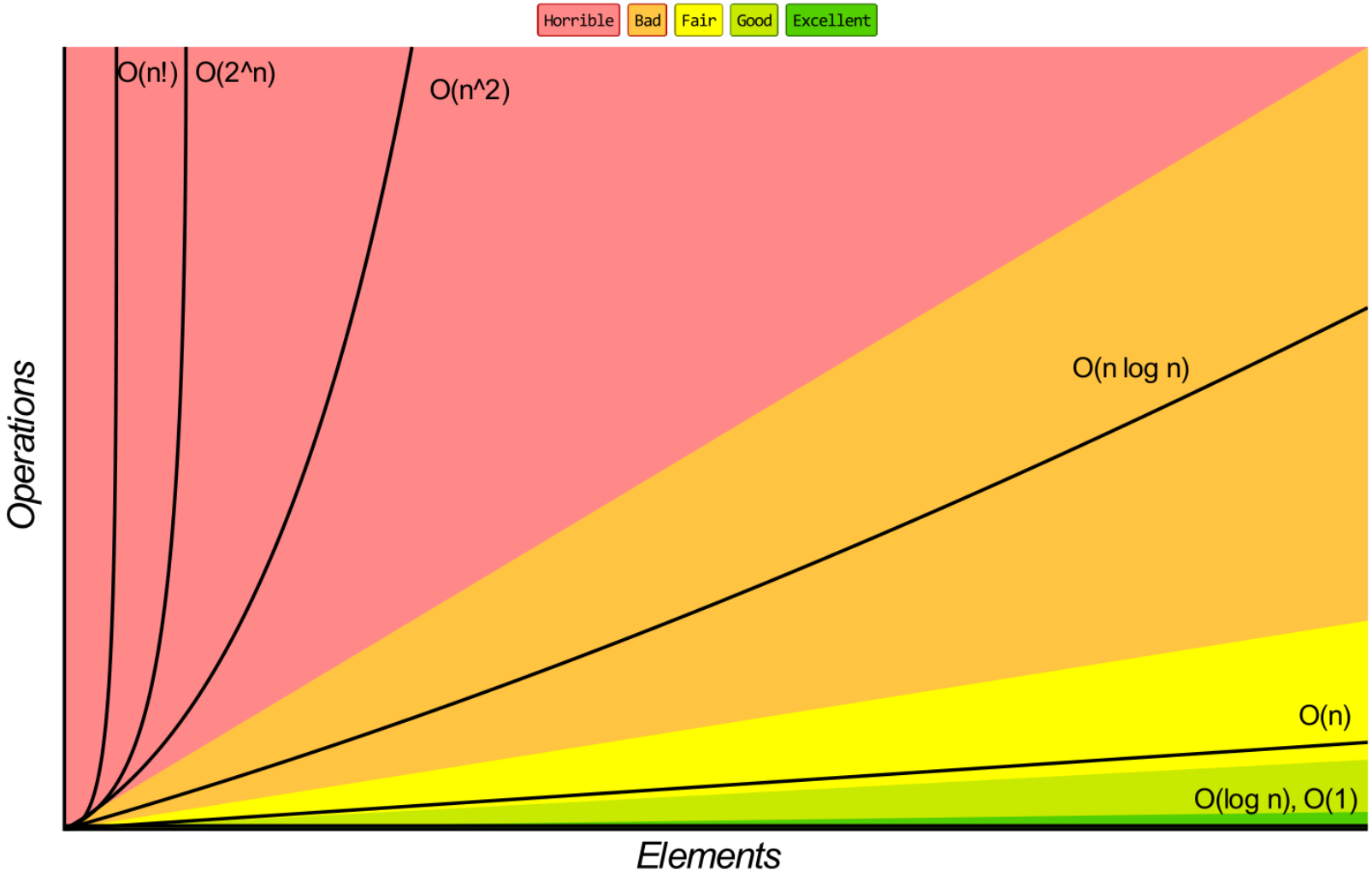

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

lst = ListNode(3)
b = ListNode(4)
c = ListNode(1)
lst.next = b
b.next = c
```

Operation	Example	Time
Construct	A = [3, 4, …]	O(n)
Access	A[5]	O(n)
Insert	node.insert(6)	O(1)
Delete	node.delete()	O(1)
Search	if 9 in A	O(n)

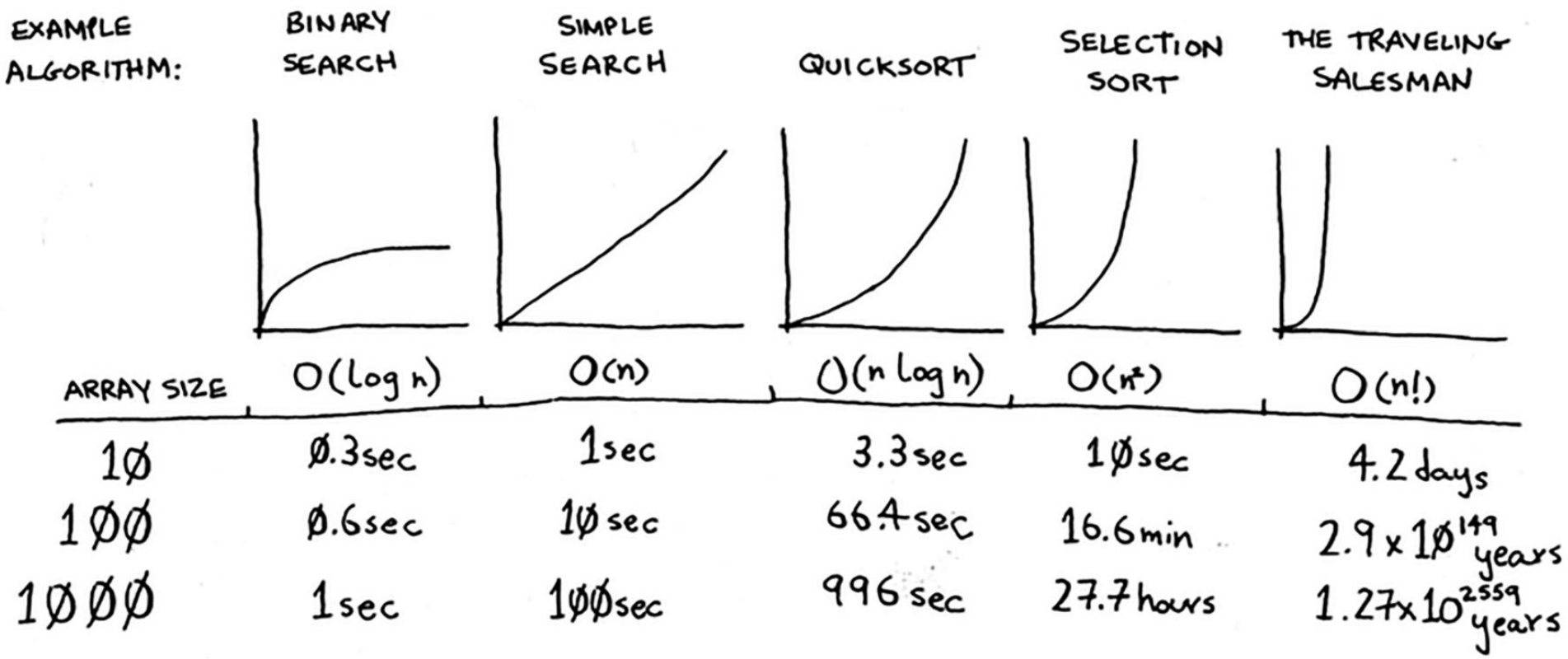
Know Thy Complexities!

www.bigocheatsheet.com

Big-O Complexity Chart



EXAMPLE ALGORITHM:	BINARY SEARCH	SIMPLE SEARCH	QUICKSORT	SELECTION SORT	THE TRAVELING SALESMAN
					
ARRAY SIZE	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$
10	0.3 sec	1 sec	3.3 sec	10 sec	4.2 days
100	0.6 sec	10 sec	664 sec	16.6 min	2.9×10^{149} years
1000	1 sec	100 sec	996 sec	27.7 hours	1.27×10^{2559} years



What if we make computer 10^{10} times faster?

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Exercise:

Implement insert/delete/access for linked list using Python

Suggestion: code it, if you cannot do it in your head

Dict, Set, Stack, Queue

dict:

D[key] = val: high probability O(1)

key in D: high probability O(1)

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["model"]

"model" in thisdict
```

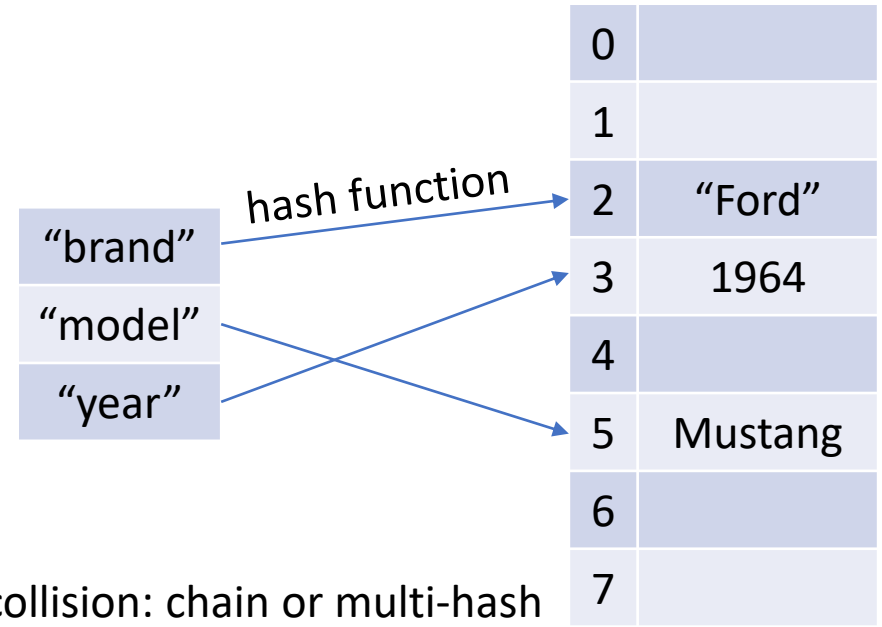
dict:

D[key] = val: high probability O(1)

key in D: high probability O(1)

probability in the sense of hash

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["model"]

"model" in thisdict
```



Key collision: chain or multi-hash

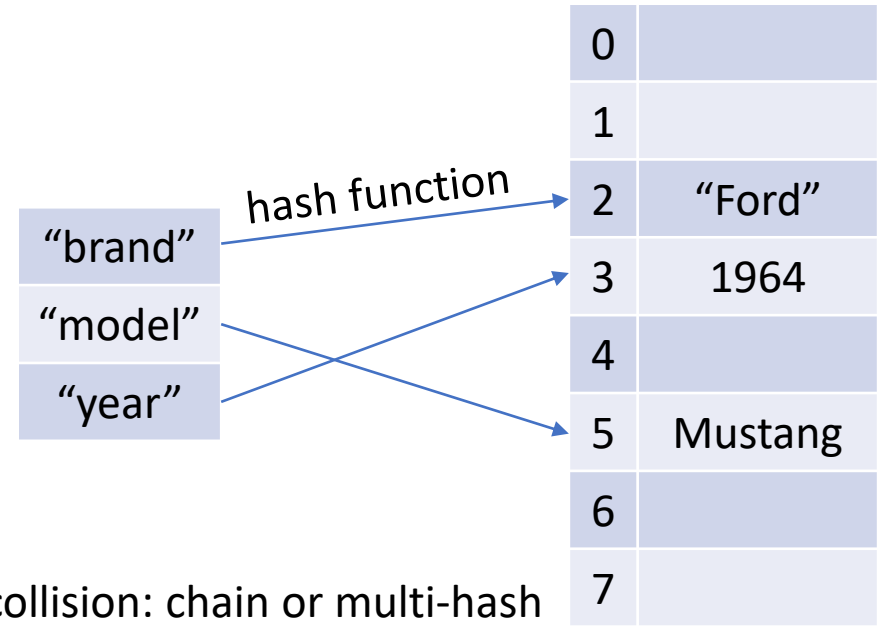
dict:

D[key] = val: high probability O(1)

key in D: high probability O(1)

probability in the sense of hash

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["model"]

"model" in thisdict
```



Key collision: chain or multi-hash

One-way function: starting point of modern encryption. https, bitcoin, etc.

dict:

D[key] = val: high probability O(1)

key in D: high probability O(1)

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

thisdict["model"]

"model" in thisdict
```

set: (dict with only key, without values)

key in D: high probability O(1)

```
thisset = {"apple", "banana", "cherry"}
"apple" in thisset
```

collections.deque: doubly linked list for queue, stack

```
from collections import deque
d = deque('ghi')
d.append('j') # ['g', 'h', 'i', 'j']
d.appendleft('f') # ['f', 'g', 'h',
'i', 'j']
d.pop() # ['f', 'g', 'h', 'i']
d.popleft() # ['g', 'h', 'i']
```

O(1)

Stack (Last In First Out, LIFO): push, pop

realization 1: list (append, pop)

realization 2: collections.deque(append, pop)

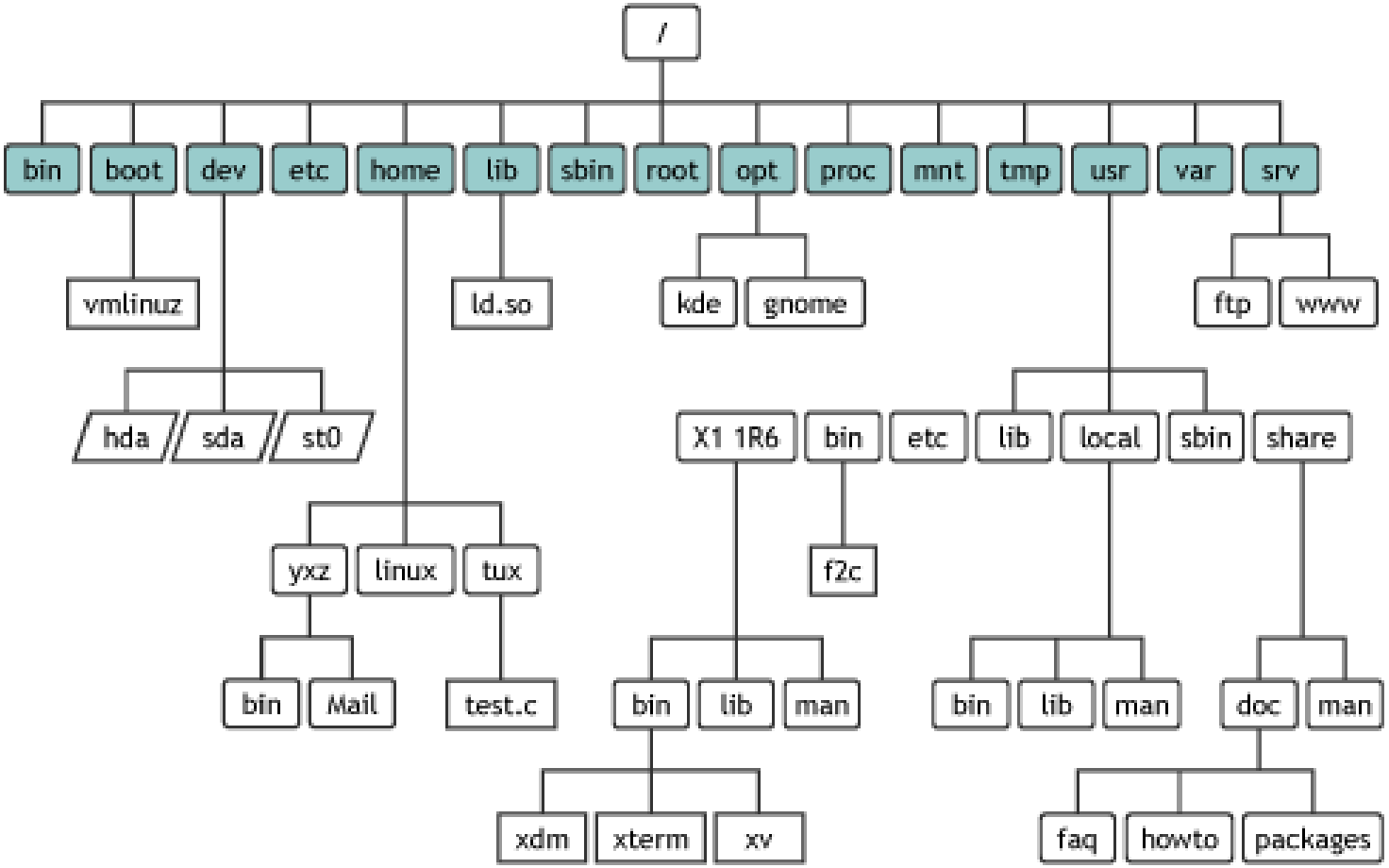
Queue (First In First Out, FIFO): enqueue, dequeue

Realization: collections.deque (appendleft, pop)

Tree, Graph

Tree data structure example: Linux file system

root is not root, / is



No child: leaf

Concepts: nodes, parent, children, level, height, path, subtree

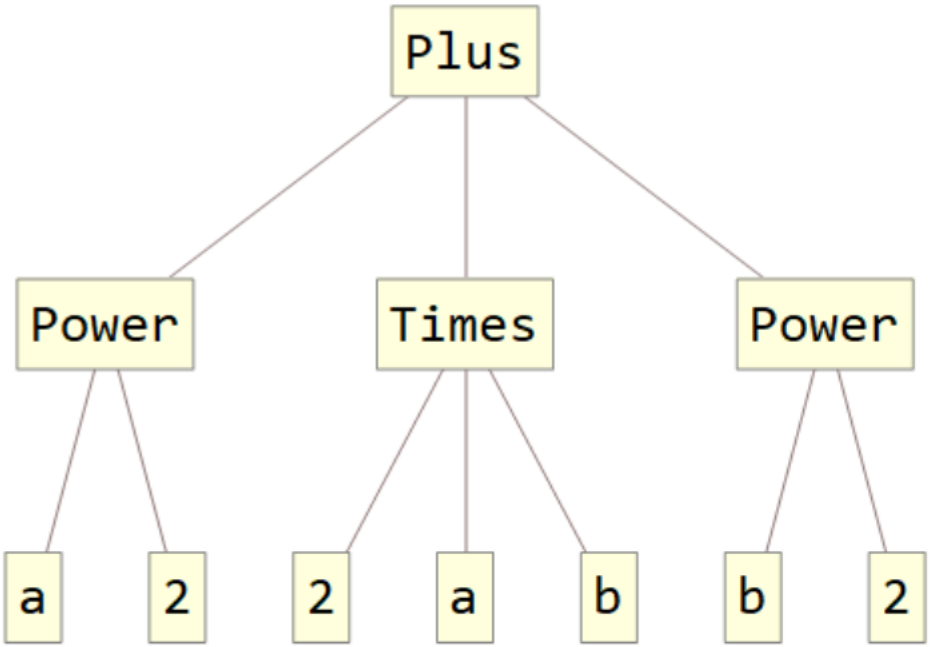
Note: No cycles in a tree (~~在地願爲連理枝~~)

Example: This is not a tree because it has cycles

Example: Expression trees

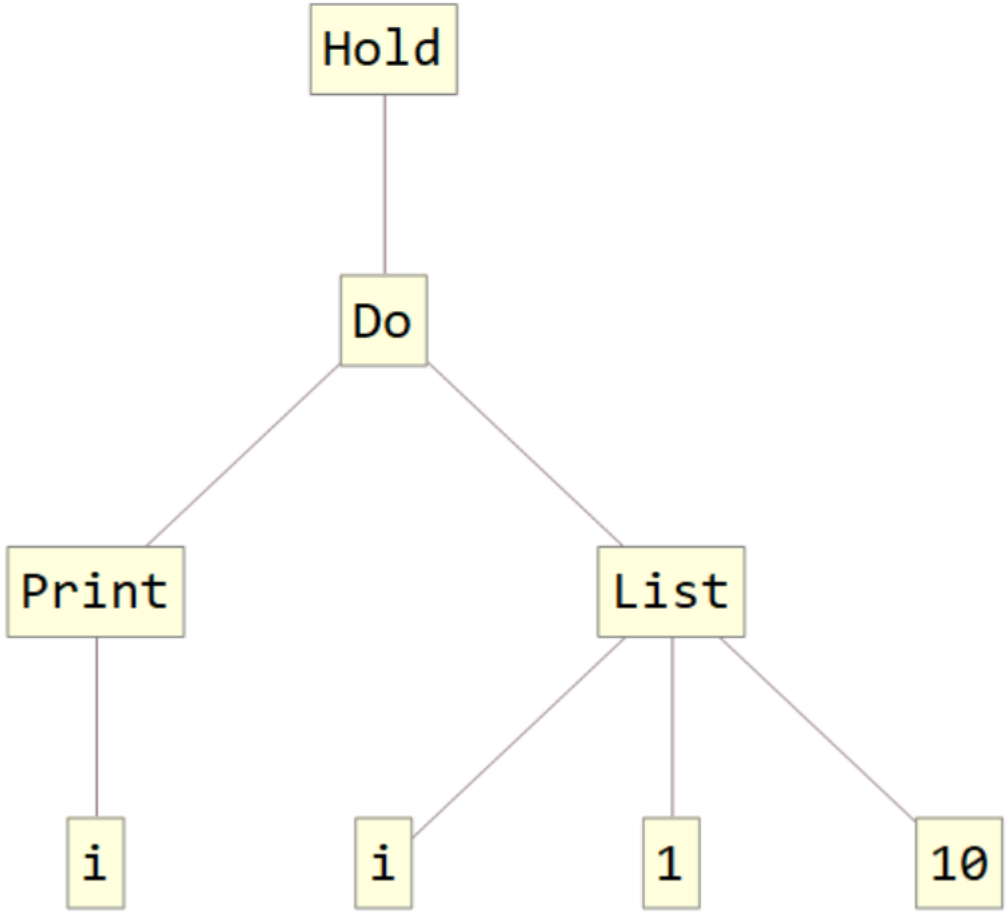
In[2]:= $a^2 + 2\,a\,b + b^2$ // TreeForm

Out[2]//TreeForm=

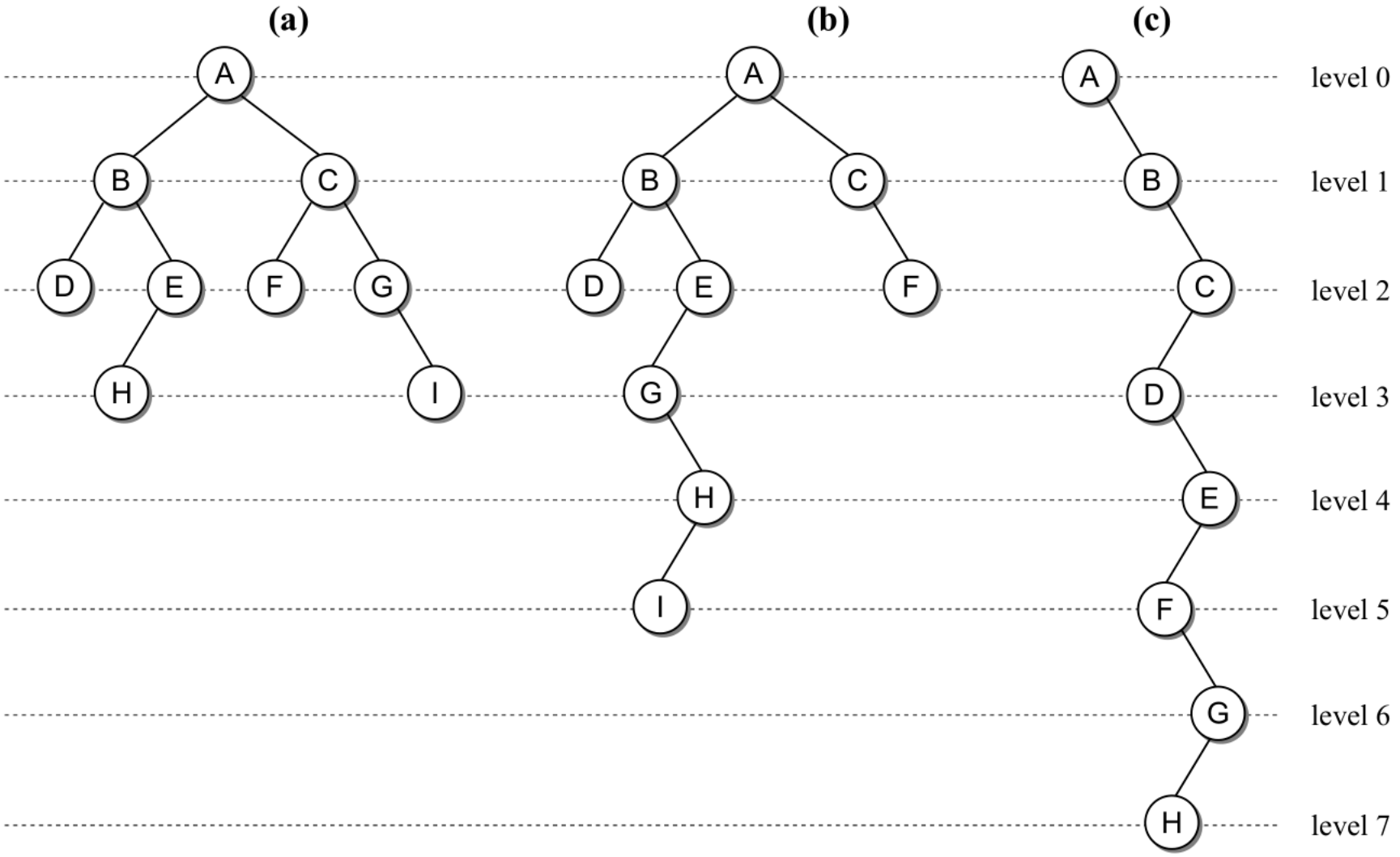


In[3]:=
```
Hold[Do[Print[i], {i, 1, 10}]] // TreeForm
```

Out[3]//TreeForm=



Binary tree: parents at most have two children



Realization of binary tree in Python

```python
class BinTreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Direct usage:
root = BinTreeNode("a")
root.left = BinTreeNode("b")
root.right = BinTreeNode("c")
root.right.right = BinTreeNode("d")
```

```python
# Tree as a class with methods:

class BinaryTree:

    def __init__(self):
        self.root = None

    def __str__(self, ...):
        ...

    def DFS(self, ...):
        ...

    def BFS(self, ...):
        ...
```

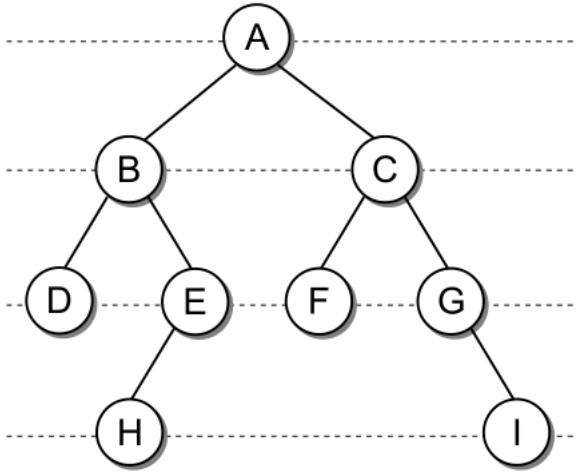
Travel through a tree, e.g.

Depth first search (DFS)

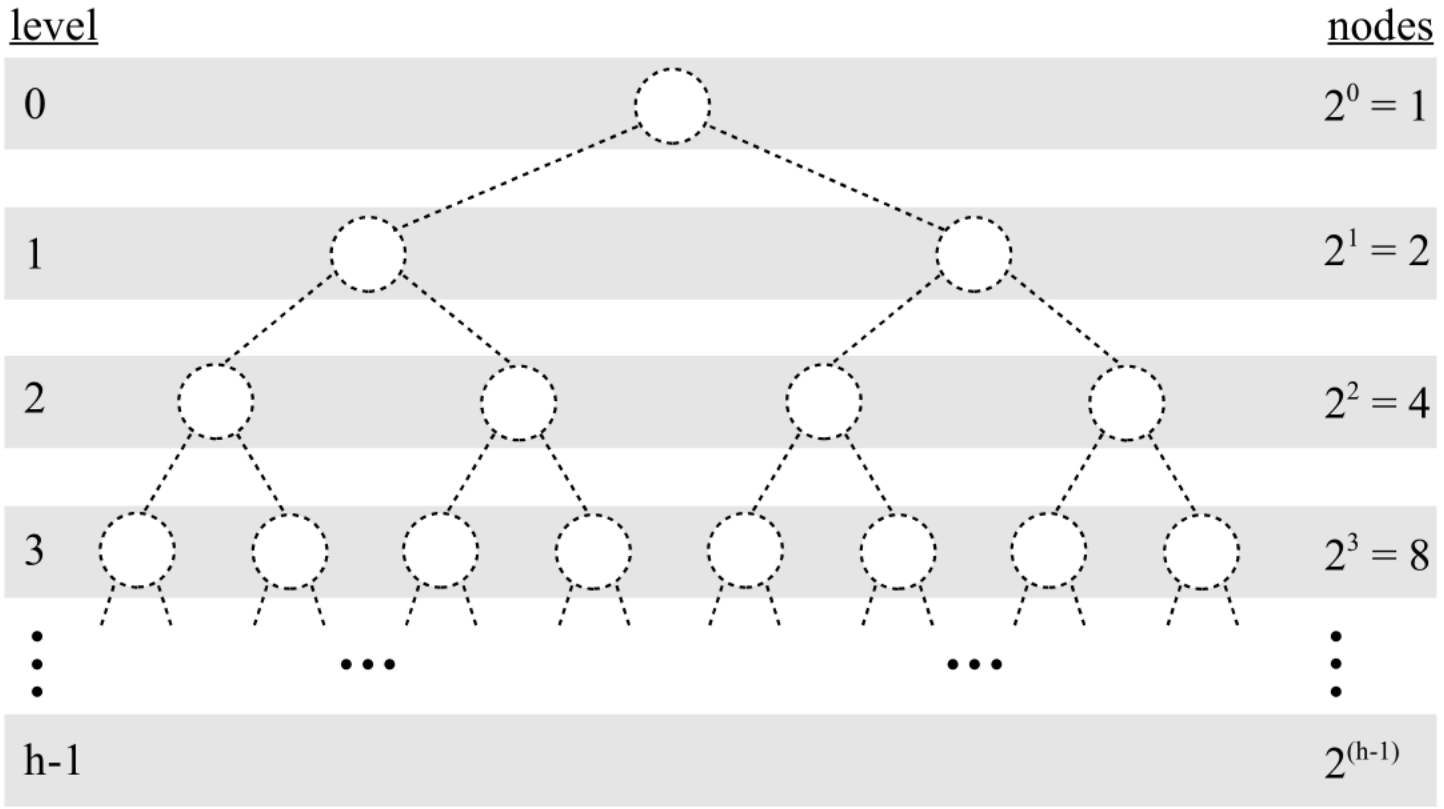
pre-order, in-order, post-order

Breadth first search (BFS)

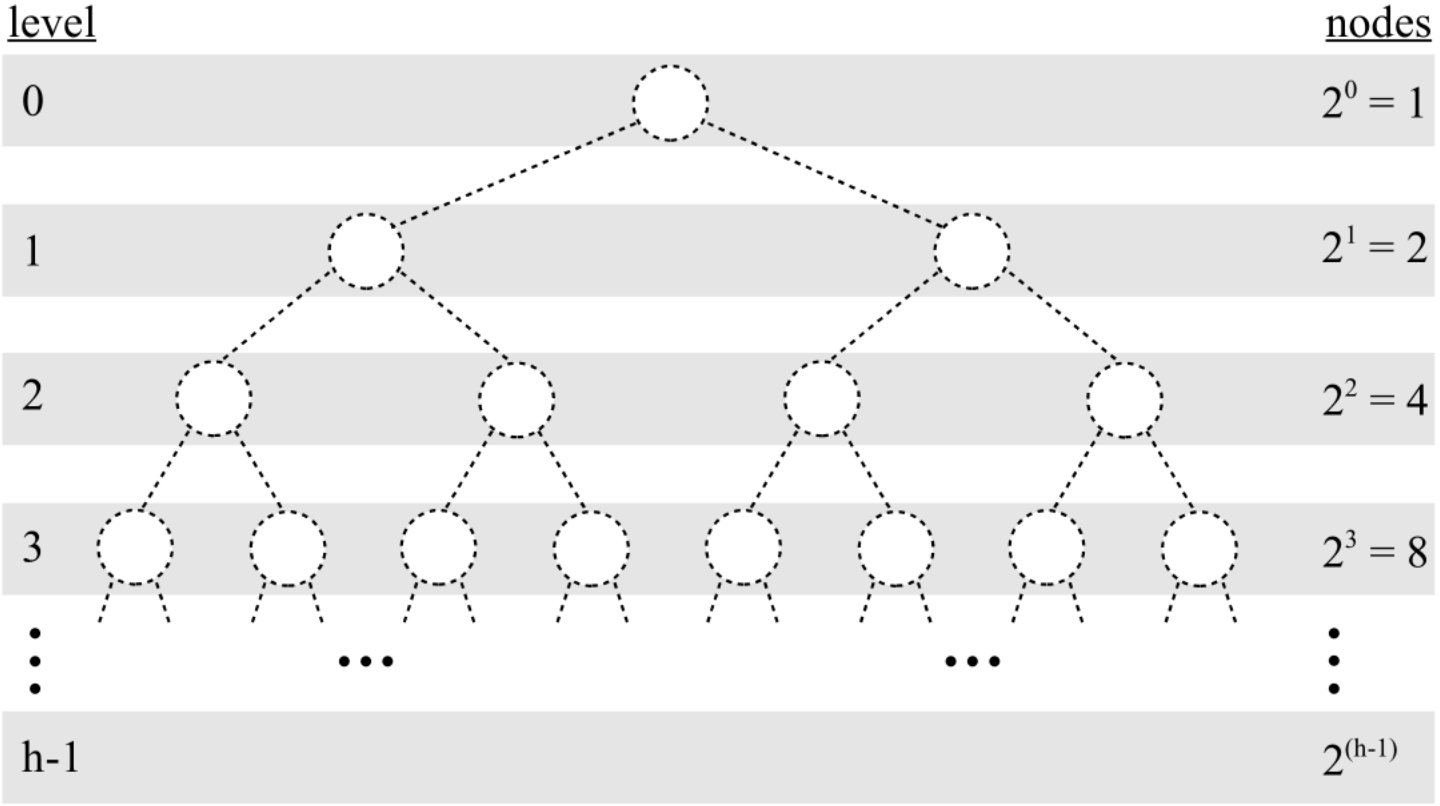
Rough ideas here. More later



What's the minimal/max height for a tree with n nodes?



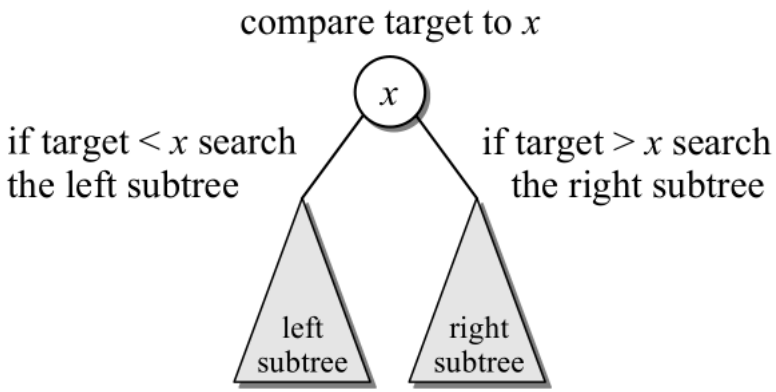
What's the minimal/max height for a tree with n nodes?

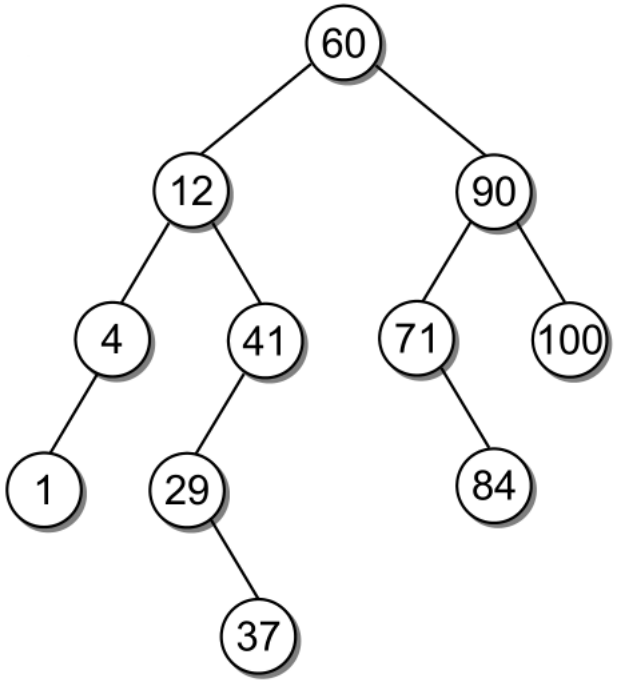


$h_{\max} = n - 1$

$h_{\min} = [\log_2 n]$ (where lots of O(logN) come from)

Example of binary trees: binary search tree



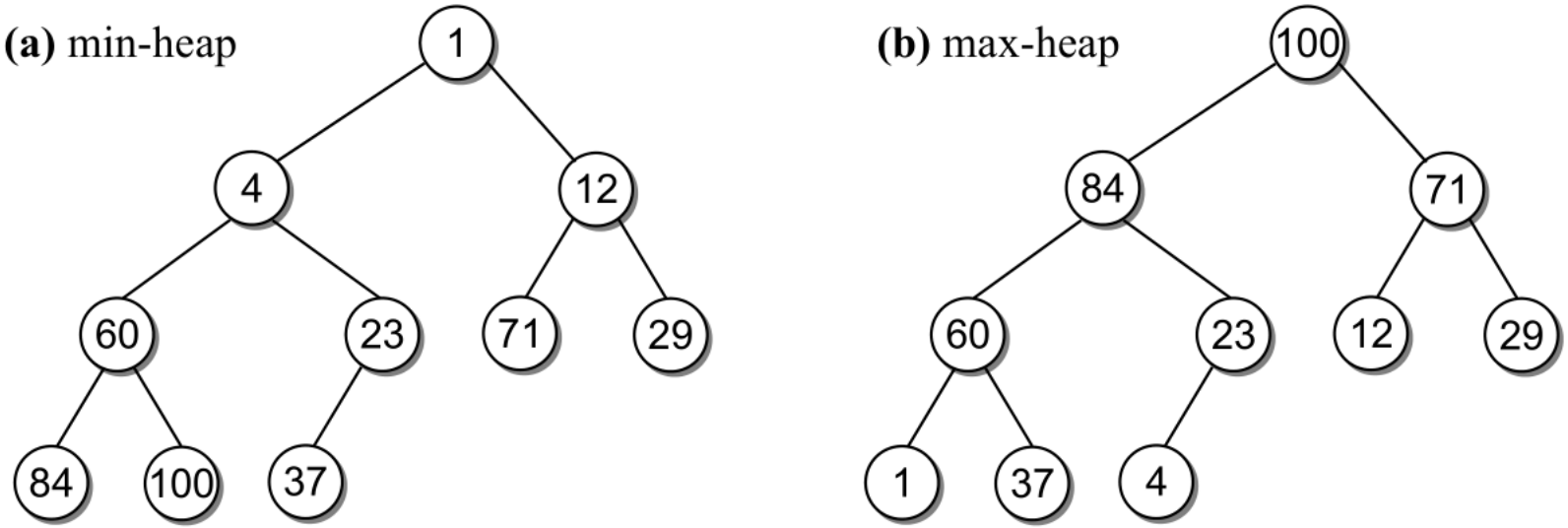


challenge: how to minimize height

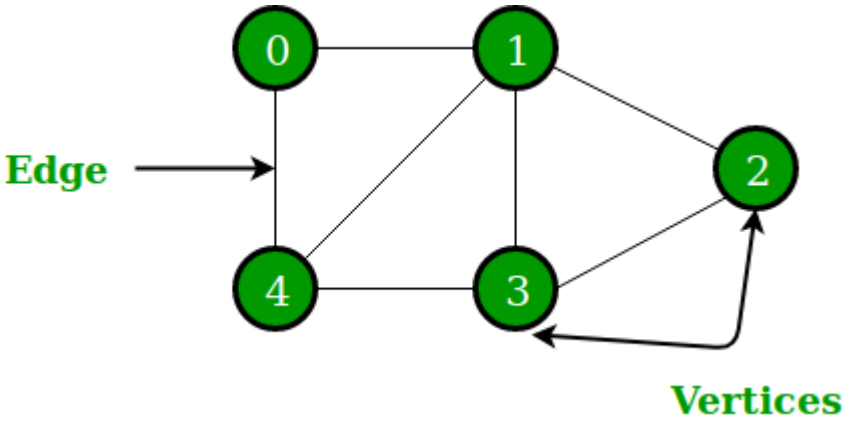
Examples of binary trees: heap

(Usually stored in a linear list)

(more later)



Graphs (more later)



Set of vertices & set of edges

Difference from trees: can have loops

Strings

(Not most relevant to this course, but you should know)

Below summary from

https://www.shortcutfoo.com/app/dojos/python-strings/cheatsheet

Sequence Operations I

s2 in s	Return true if s contains s2
s + s2	Concat s and s2
len(s)	Length of s
min(s)	Smallest character of s
max(s)	Largest character of s

Sequence Operations II

s2 not in s	Return true if s does not contain s2
s * integer	Return integer copies of s concatenated # 'hello' => 'hellohellohello'
s[index]	Character at index of s
s[i:j:k]	Slice of s from i to j with step k
s.count(s2)	Count of s2 in s

Find / Replace I

s.index(s2, i, j)	Index of first occurrence of s2 in s after index i and before index j
s.find(s2)	Find and return lowest index of s2 in s
s.index(s2)	Return lowest index of s2 in s (but raise ValueError if not found)
s.replace(s2, s3)	Replace s2 with s3 in s
s.replace(s2, s3, count)	Replace s2 with s3 in s at most count times
s.rfind(s2)	Return highest index of s2 in s
s.rindex(s2)	Return highest index of s2 in s (raise ValueError if not found)

Inspection I

s.endswith(s2)	Return true if s ends with s2
s.isalnum()	Return true if s is alphanumeric
s.isalpha()	Return true if s is alphabetic
s.isdecimal()	Return true if s is decimal
s.isnumeric()	Return true if s is numeric
s.startswith(s2)	Return true is s starts with s2

Inspection II

s.endswith((s1, s2, s3))	Return true if s ends with any of string tuple s1, s2, and s3
s.isdigit()	Return true if s is digit
s.isidentifier()	Return true if s is a valid identifier
s.isprintable()	Return true is s is printable

Splitting I

s.join('123')	Return s joined by iterable '123' # 'hello' => '1hello2hello3'
s.partition(sep)	Partition string at sep and return 3-tuple with part before, the sep itself, and part after # 'hello' => ('he', 'l', 'lo')
s.rpartition(sep)	Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after # 'hello' => ('hel', 'l', 'o')
s.rsplit(sep, maxsplit)	Return list of s split by sep with rightmost maxsplits performed
s.split(sep, maxsplit)	Return list of s split by sep with leftmost maxsplits performed
s.splitlines()	Return a list of lines in s # 'hello\nworld' => ['hello', 'world']

Whitespace I

s.center(width)	Center s with blank padding of width # 'hi' => ' hi '
s.isspace()	Return true if s only contains whitespace characters
s.ljust(width)	Left justifiy s with total size of width # 'hello' => 'hello '
s.rjust(width)	Right justify s with total size of width # 'hello' => ' hello'
s.strip()	Remove leading and trailing whitespace from s # ' hello ' => 'hello'

Whitespace II

s.center(width, pad)	Center s with padding pad of width # 'hi' => 'padpadhipadpad'
s.expandtabs(integer)	Replace all tabs with spaces of tabsize integer # 'hello\tworld' => 'hello world'
s.lstrip()	Remove leading whitespace from s # ' hello ' => 'hello '
s.rstrip()	Remove trailing whitespace from s # ' hello ' => ' hello'
s.zfill(width)	Left fill s with ASCII '0' digits with total length width # '42' => '00042'

Cases I

s.capitalize()	Capitalize s # 'hello' => 'Hello'
s.lower()	Lowercase s # 'HELLO' => 'hello'
s.swapcase()	Swap cases of all characters in s # 'Hello' => "hELLO"
s.title()	Titlecase s # 'hello world' => 'Hello World'
s.upper()	Uppercase s # 'hello' => 'HELLO'

Cases II

s.casefold()	Casefold s (aggressive lowercasing for caseless matching) # 'ßorat' => 'ssorat'
s.islower()	Return true if s is lowercase
s.istitle()	Return true if s is titlecased # 'Hello World' => true
s.isupper()	Return true if s is uppercase

I-b: Introduction to Algorithms

Systematic ways to tell the computer what to do, and how to do it.

Iteration:

```
for i in range(10):
    print(i)
```

What’s the output?

Iteration:

```
for i in range(10):
    print(i)

for c in "Hello World!":
    print(c)
```

Iterators:
tuple: (1,2,3)
list: [1,2,3]
set: {1,2,3}
dict: {‘a’:1,’b’:2}
string: ‘abc’
write your own

Comprehension (list, tuple, set, …)

```
[i**2 for i in range(10) if i%2 == 0]
```

See also while

Iteration example 1: two-sum

Given number R, and a list A = {A[0], A[1], …, A[N-1]},

find two numbers in A which sums to R,

i.e. find i, j, s.t. A[i] + A[j] = R,

or tell it’s impossible if it’s the case.

Iteration example 2: find largest sum of subsequences

Given a list A = {A[0], A[1], … A[N-1]}, find the largest A[i] + … + A[j] is among all possible choices of i & j.

Iteration example 2: find largest sum of subsequences

Given a list A = {A[0], A[1], … A[N-1]}, find the largest A[i] + … + A[j] is among all possible choices of i & j.

```
1  def max_subarray(numbers):
2      """Find the largest sum of any contiguous subarray."""
3      best_sum = 0  # or: float('-inf')
4      current_sum = 0
5      for x in numbers:
6          current_sum = max(0, current_sum + x)
7          best_sum = max(best_sum, current_sum)
8      return best_sum
```

Recursion: What’s recursion?

（~~递~~~~龟~~递归）

Recursion: What's recursion?

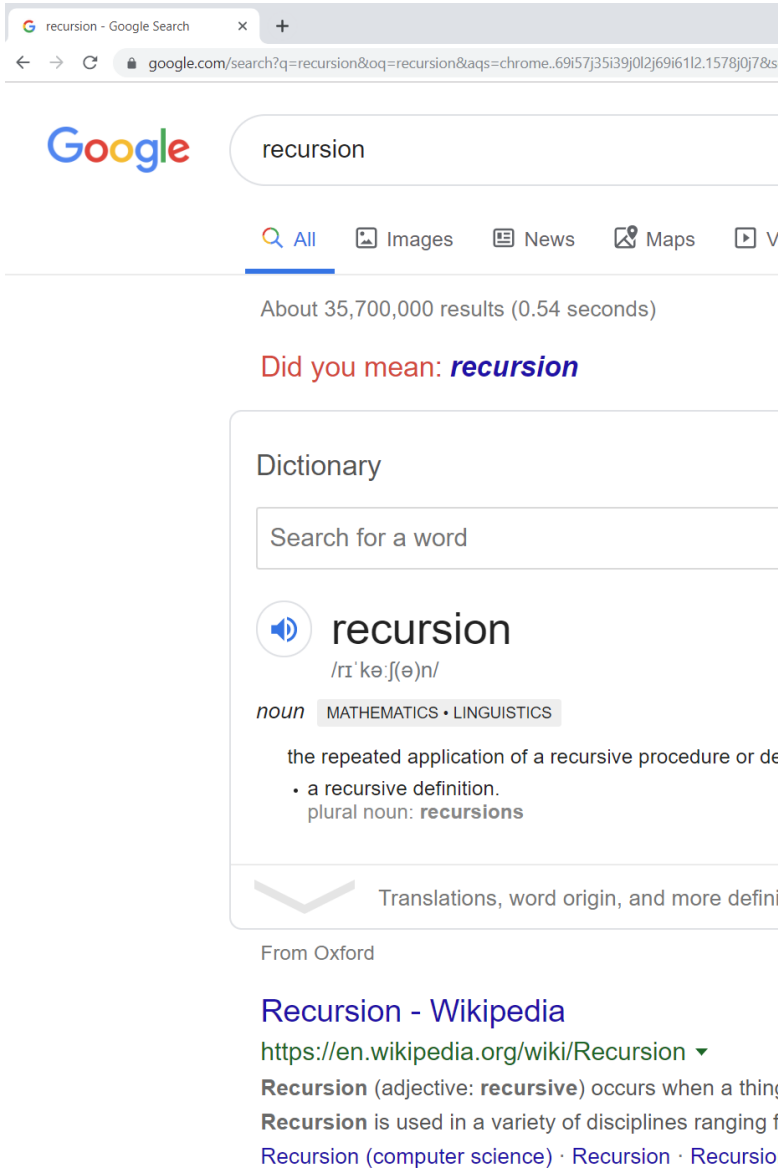
It's said that,

to understand recursion,

you have to first understand recursion.

Recursion: What's recursion?

See what Google tells:



Recursion: a function call itself.

從前有座山，山裏有個廟，廟裏有個老和尚給小和尚講故事:

“從前有座山，山裏有個廟，廟裏有個老和尚給小和尚講故事:

“從前有座山，山裏有個廟，廟裏有個老和尚給小和尚講故事:

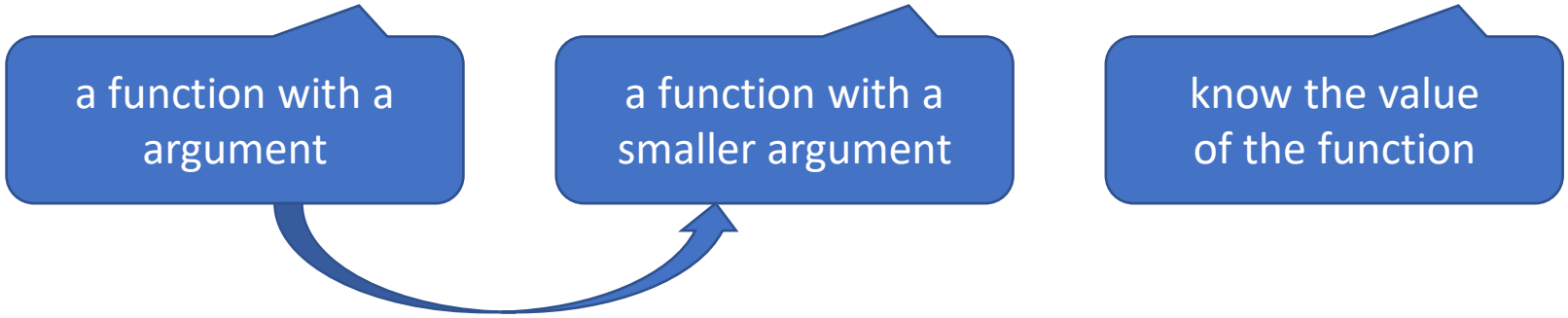
……””

Why a function likes to call itself?

One of the most important idea in programming:

Divide-and-conquer

Divide a problem into subproblems, until simple enough



call recursively

exit condition

How a programmer braid her hair?

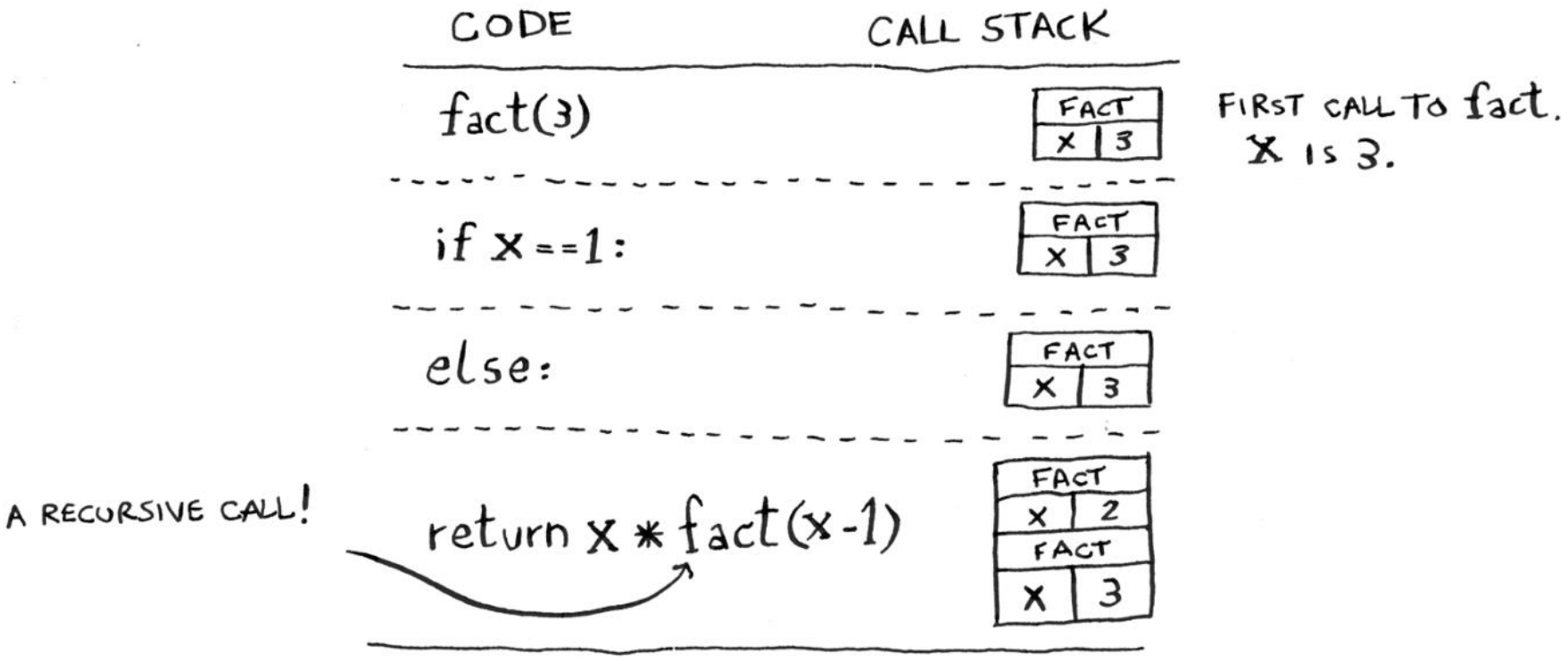
How a programmer braid her hair?

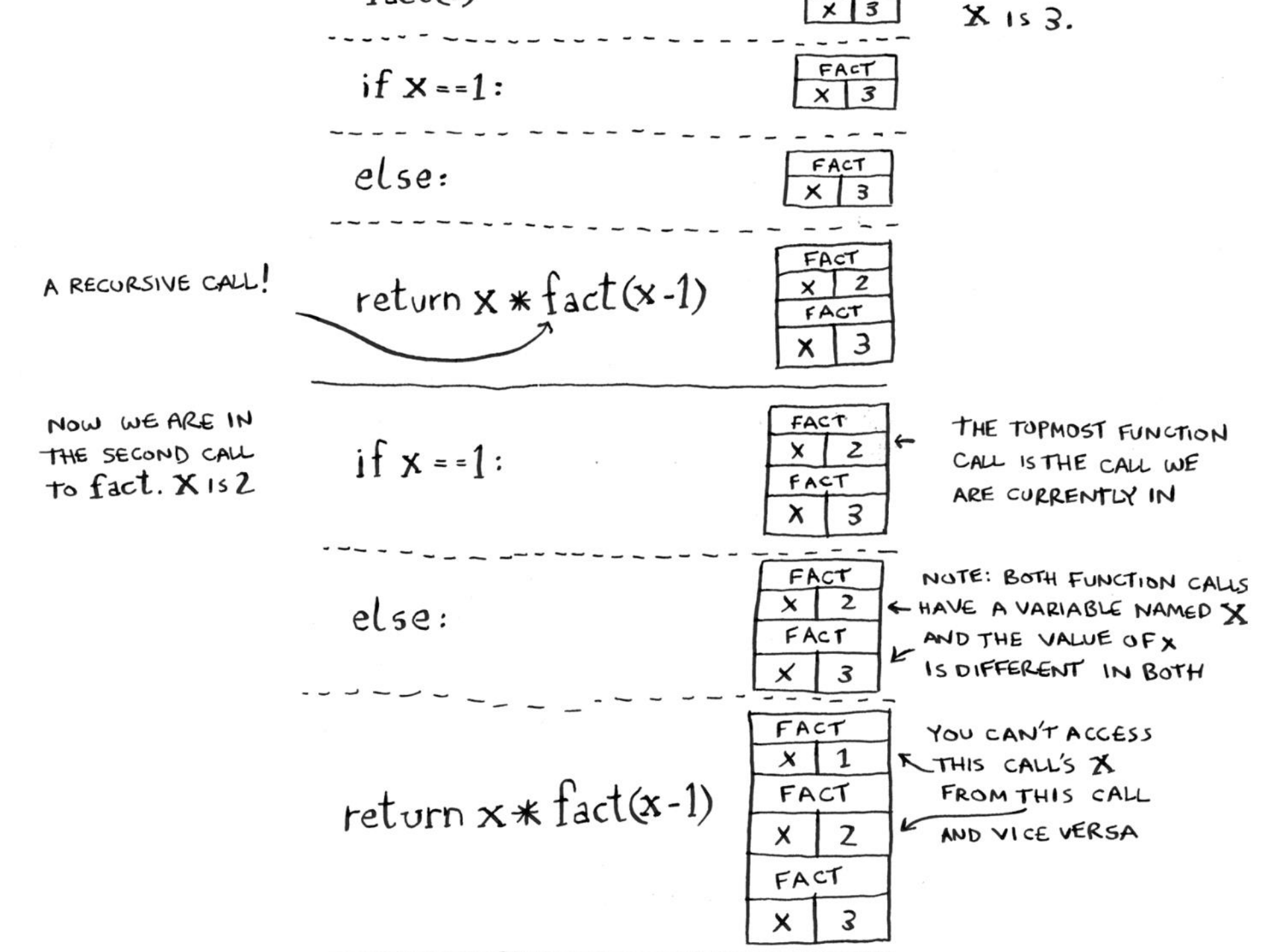
Divide-and-conquer

Recursion example 1: factorial

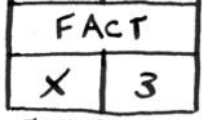
```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(6) # 720
```

Explanation about factorial (fact for short) in recursion and how call stacks work (from Grokking Algorithms)



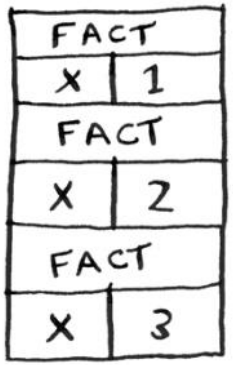

X | 3
X is 3.
if x==1:
FACT
X | 3
else:
FACT
X | 3
A RECURSIVE CALL!
return x * fact(x-1)
FACT
X | 2
FACT
X | 3
NOW WE ARE IN THE SECOND CALL to fact. X is 2
if x==1:
FACT
X | 2
FACT
X | 3
THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN
else:
FACT
X | 2
FACT
X | 3
NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF x IS DIFFERENT IN BOTH
return x * fact(x-1)
FACT
X | 1
FACT
X | 2
FACT
X | 3
YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA

else:



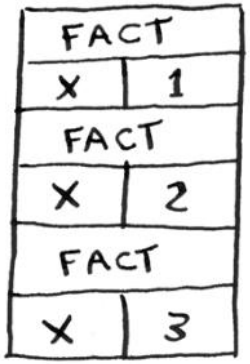
AND THE VALUE OF x IS DIFFERENT IN BOTH

return x * fact(x-1)

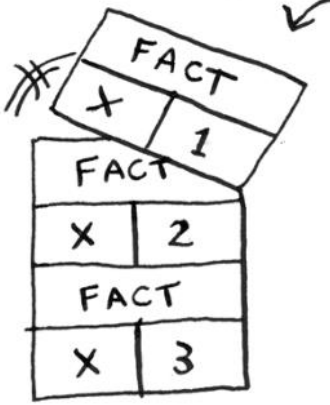


YOU CAN'T ACCESS THIS CALL'S x FROM THIS CALL AND VICE VERSA

if x == 1:



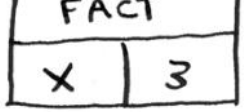
return 1



THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS ITS THE FIRST CALL WE RETURN FROM

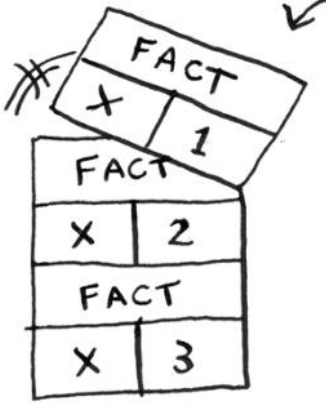
RETURNS 1

WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!



WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!

return 1



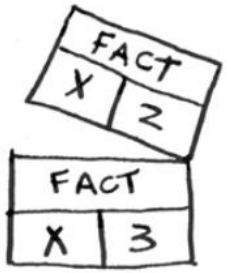
THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS ITS THE FIRST CALL WE RETURN FROM

← RETURNS 1

THIS IS THE FUNCTION CALL WE JUST RETURNED FROM

return x * fact(x-1)

x is 2

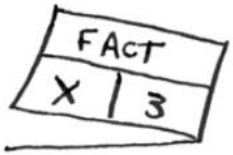


← RETURNS 2

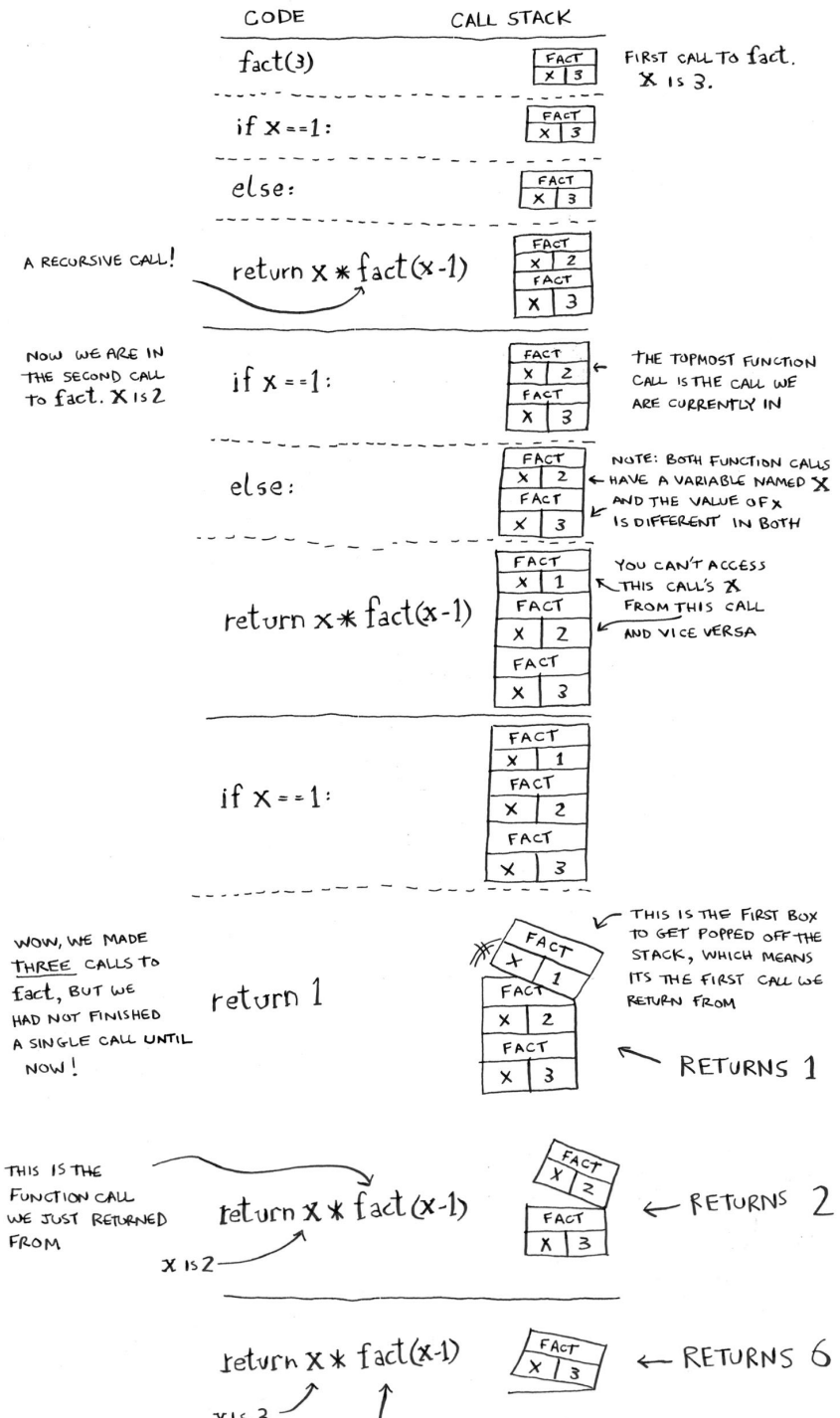
return x * fact(x-1)

x is 3

THIS CALL RETURNED 2



← RETURNS 6


CODE
CALL STACK
fact(3)
FACT x 3
FIRST CALL TO fact. X IS 3.
if x==1:
else:
A RECURSIVE CALL!
return x * fact(x-1)
FACT x 2
FACT x 3
NOW WE ARE IN THE SECOND CALL to fact. X IS 2
THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN
NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF x IS DIFFERENT IN BOTH
FACT x 1
YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA
WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!
return 1
THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS ITS THE FIRST CALL WE RETURN FROM
RETURNS 1
THIS IS THE FUNCTION CALL WE JUST RETURNED FROM
X IS 2
RETURNS 2
RETURNS 6

push

pop

Recursion example 1: factorial

```python
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(6) # 720
```

Pros:

1. A little shorter

How recursion works? Stack realized by system

Recursion example 1: factorial

```python
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(6) # 720
```

Pros:

1. A little shorter

Why don’t we use iteration?

Recursion example 1: factorial

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(6) # 720
```

Pros:

1. A little shorter

Why don't we use iteration?

```
def factorial(n):
    prod = 1
    for i in range(2,n+1):
        prod = prod * i
    return prod

factorial(6) # 720
```

Pros:

1. No function call overhead
2. Load loop var to register
   (may not be in Python)
3. Similar to what human do

Recursion example 1: factorial

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(6) # 720
```

Pros:

1. A little shorter

Why don't we use iteration?

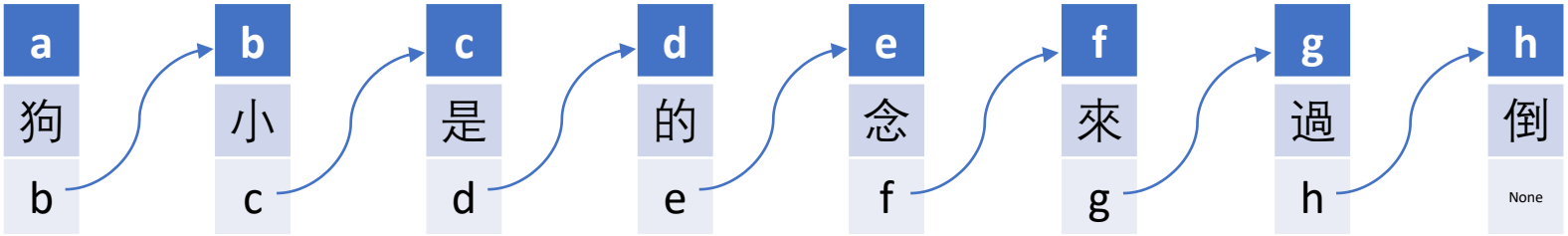
No particular reason for this example.

But recursion is more powerful in many other ways.

All recursion can be done in iteration.
But recursion versions can be more intuitive once you are familiar with it.

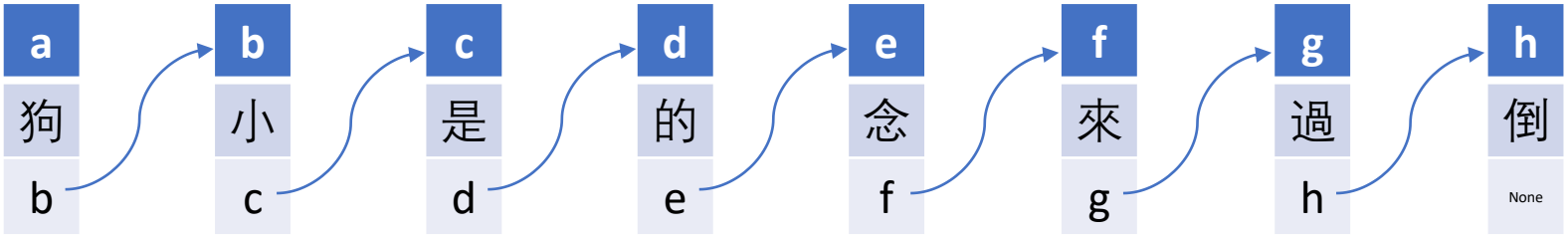
Example: reversed print of a (single) linked list

Input: (given the linked list and its head a)



Example: reversed print of a (single) linked list

Input: (given the linked list and its head a)



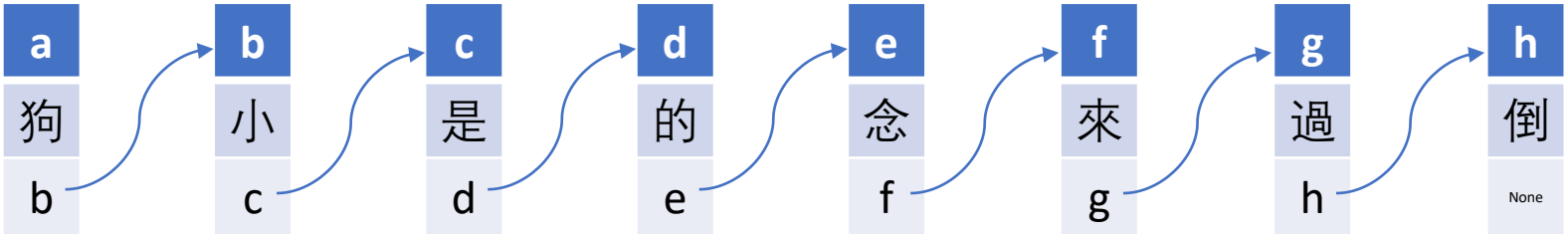
Expected output: 倒過來念的是小狗

Naïve approach: O(n^2). Better ideas?

- Iterative approach
- Recursion

Example: reversed print of a (single) linked list

Input: (given the linked list and its head a)



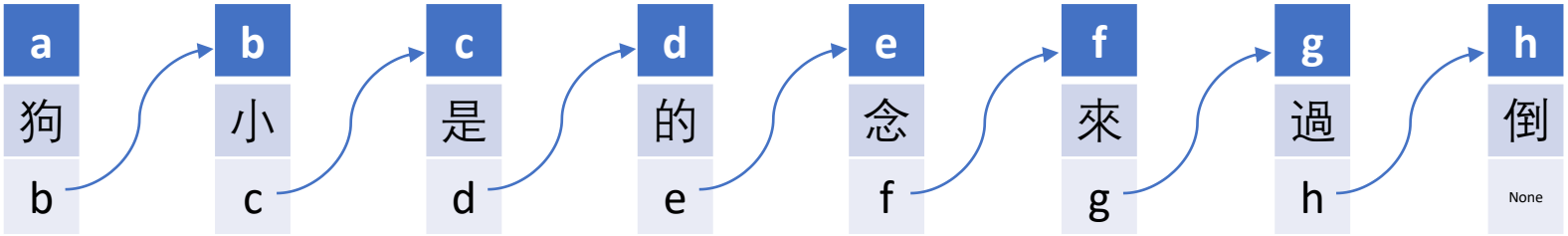
Expected output: 倒過來念的是小狗

Naïve approach: O(n^2). Better ideas?

- Iterative approach: save obj ids in an array, reverse the array
- Recursion

Example: reversed print of a (single) linked list

Input: (given the linked list and its head a)



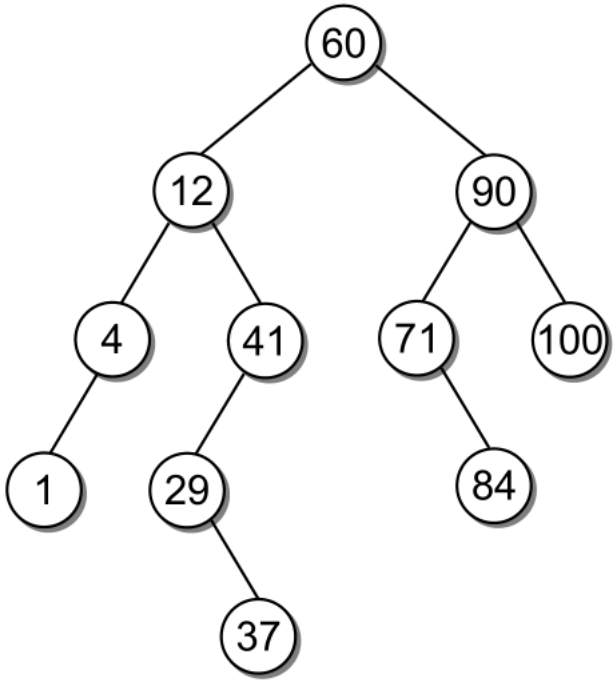
Expected output: 倒過來念的是小狗

Naïve approach: O(n^2). Better ideas?

- Iterative approach: save obj ids in an array, reverse the array
- Recursion

```
def reverse(head):
    if head is not None:
        reverse(head.next)
        print(head.data)
```

Example: print tree data by pre-order walk



Example: print tree data by pre-order walk

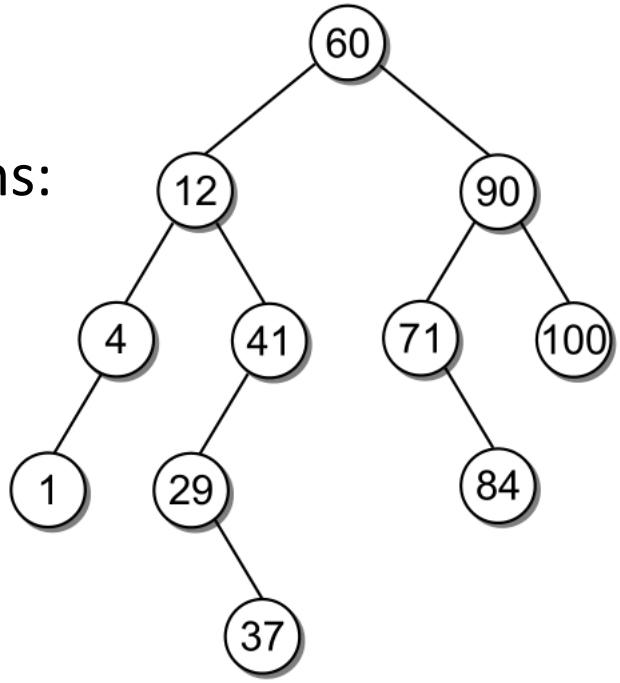
Idea: divide the problem into sub-problems:

Print node,

walk left sub-tree,

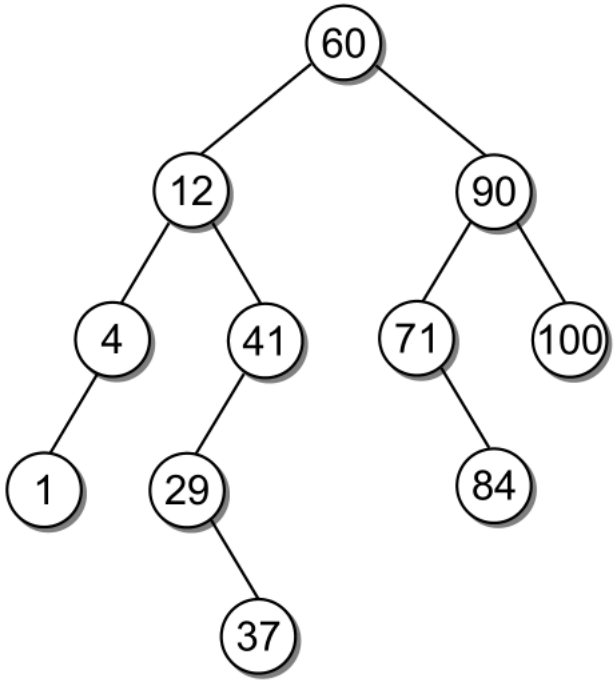
walk right sub-tree

End condition: Children is None



Example: print tree data by pre-order walk

```
def pre-order(node):
    if node:
        print(node.data)
        pre-order(node.left)
        pre-order(node.right)
```



Simple, isn't it? How it works?

You can easily make branches in recursion

(Each level corresponds to an iteration)

```
for level 0:
    for level 1:
        for level 2:
            ...
```

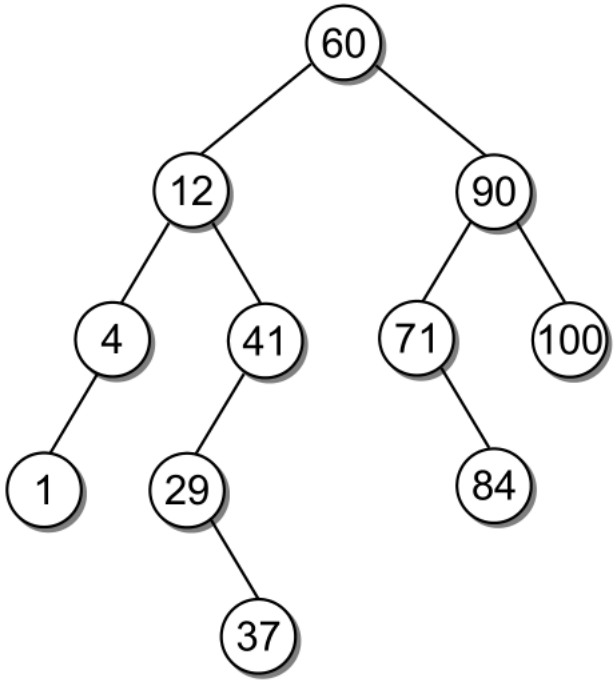
Naïve correspondence of iteration

But you don't know how many levels

(Though you can realize stack yourself)

Example: print tree data by pre-order walk

```
def pre-order(node):
    if node:
        print(node.data)
        pre-order(node.left)
        pre-order(node.right)
```

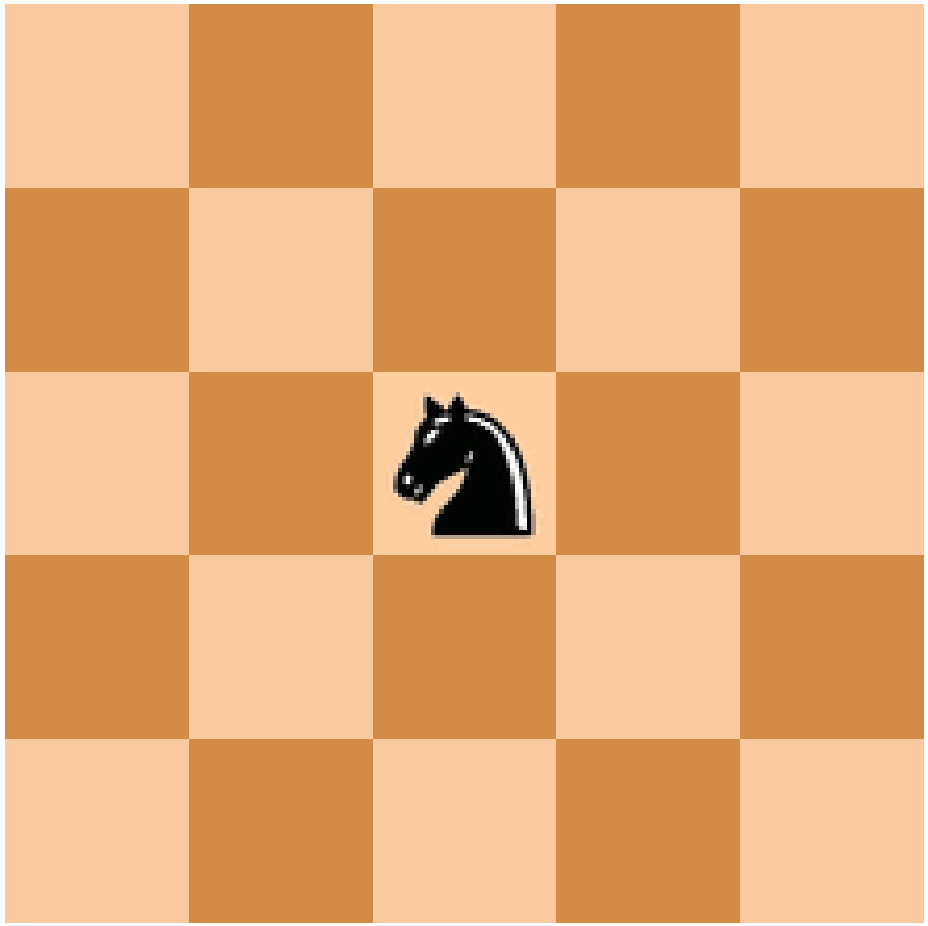


Simple, isn't it? How it works?

Exercise:

- Draw the call stacks for a small tree
- Return a list, instead of print the data
- Add pre-order as a method of class BinaryTree
- In-order and post-order

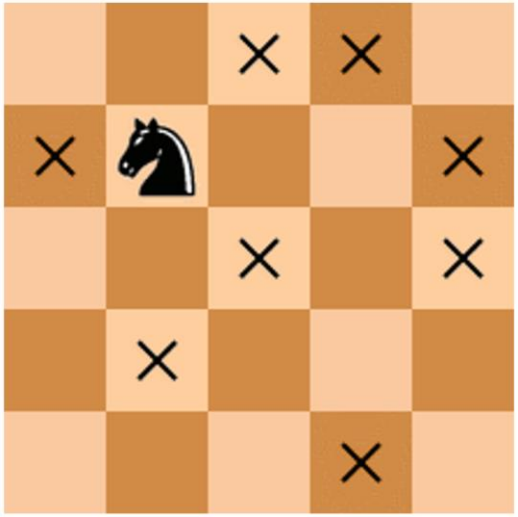
Example: Knight's tour



Question 1: How to model this problem?

≃ what data to store?

- Store horse trajectory? (hard to check availability)
- Store availability on board? (information of steps lost)
- Store steps on board (satisfactory)



Question 2: Algorithm? Recursion

- If completed: print and quit
- Mark current move on board
- Try next step (all possibilities)
- Remove current move on board

Question 3: Optimizations?

- Try complete a part before leaving (least possibilities first) [Greedy algorithm]

Question 4: Make it easier to understand, maintain & generalize

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

```
Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52
```

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
where to start?
idea -> small functions -> main

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

```
Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52
```

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
Small functions.
“Do one thing and do it well”

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
consider to use default arguments to simplify user input

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52

```
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
Names of the functions:
Easy to read, meaningful names

```
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

```
Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52
```

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
Numbers start from 0 or 1?
Following Python counting

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52

```
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
What to hard-code
and what to make variable?

```
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

```
Found solution:

0       3       60      19      40      5       42      21
33      18      1       4       59      20      39      6
2       63      34      61      36      41      22      43
17      32      47      56      45      58      7       38
48      13      62      35      54      37      44      23
31      16      55      46      57      26      53      8
12      49      14      29      10      51      24      27
15      30      11      50      25      28      9       52
```

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
moves, or move_x, move_y?
get_next_position or get_next_move?

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
global vs local variables?

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

Design consideration:
Object oriented or not?

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

```
Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52
```

```python
def move_available(position, move):
    x = position[0] + move[0]
    y = position[1] + move[1]
    return 0 <= x < size[0] and 0 <= y < size[1] and board[x][y] == -1

def get_next_positions(position):
    moves = [[1,2], [2,1], [1,-2], [2,-1], [-1,2], [-2,1], [-1,-2], [-2,-1]]
    return [[position[0] + move[0], position[1] + move[1]] for move in moves if move_available(position, move)]

def get_next_positions_sorted_by_least_next_moves(position):
    return sorted(get_next_positions(position), key=lambda next_position: len(get_next_positions(next_position)))

def print_chess_board_state():
    print("\nFound solution: \n")
    for x in range(size[0]):
        print(*board[x],'\n',sep='\t')

def horse_move(position, step=0):
    board[position[0]][position[1]] = step
    if step == size[0] * size[1] - 1:
        print_chess_board_state()
        quit()
    for next_position in get_next_positions_sorted_by_least_next_moves(position):
        horse_move(next_position, step + 1)
    board[position[0]][position[1]] = -1
```

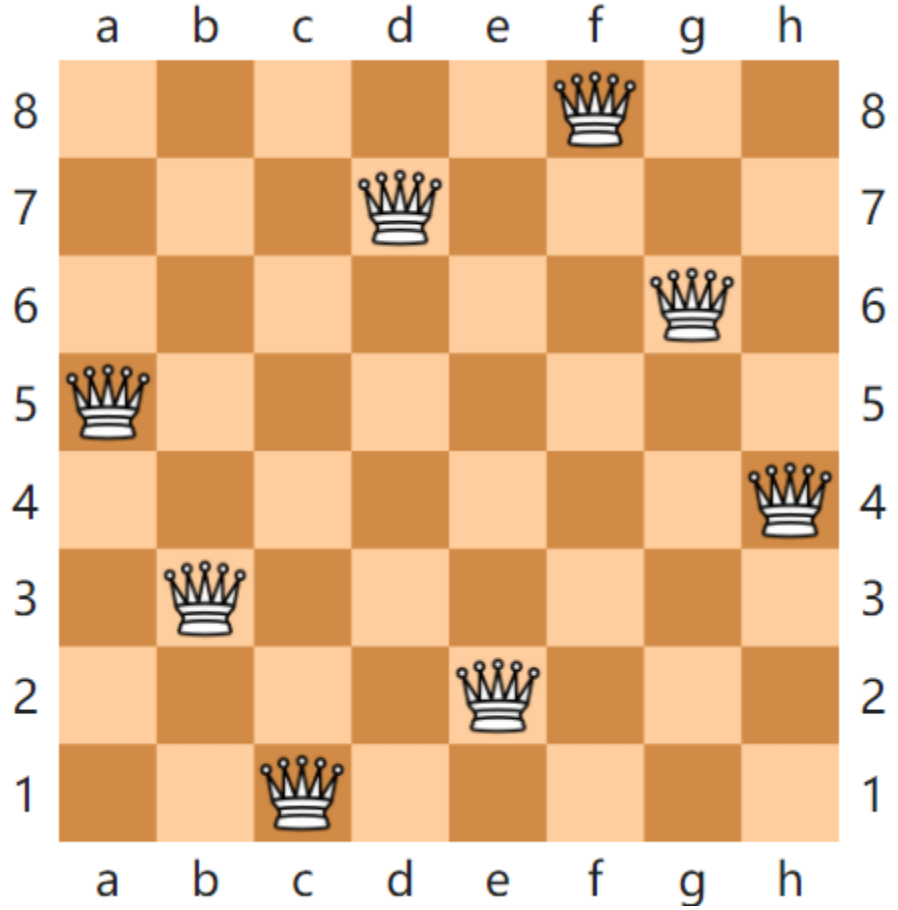
Design consideration:
My lines are sometimes too long, sorry …

```python
size = [8,8]
board = [[-1 for i in range(size[0])] for j in range(size[1])]
horse_move([0,0])
```

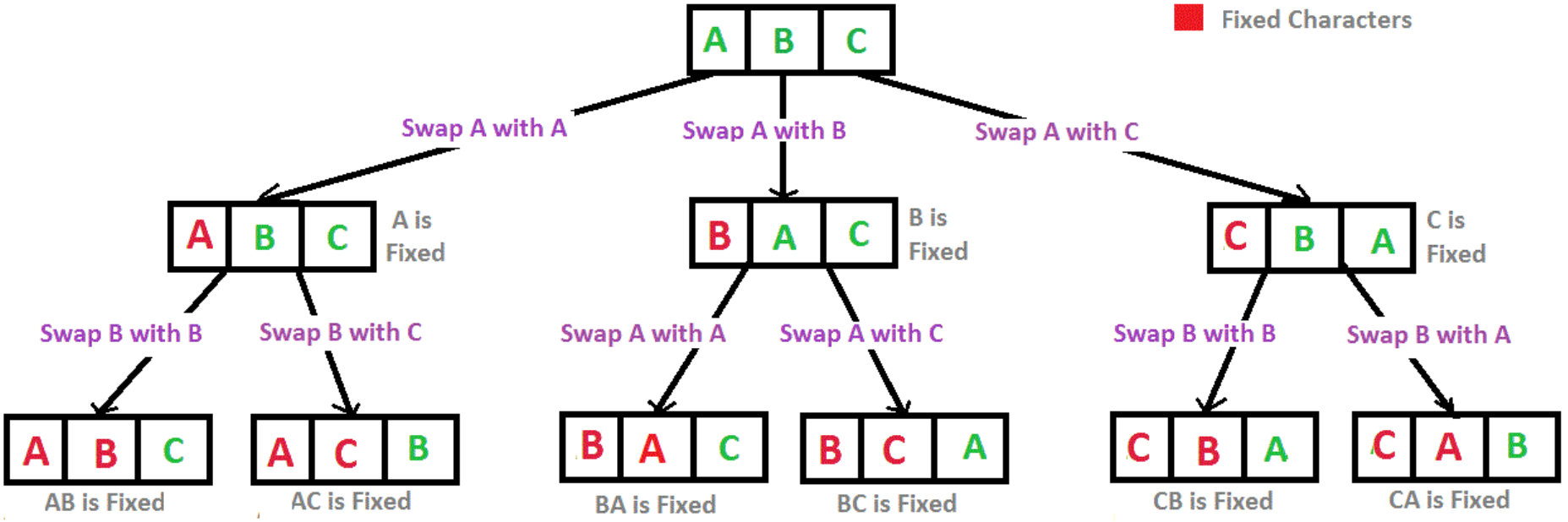
```
Found solution:

0	3	60	19	40	5	42	21
33	18	1	4	59	20	39	6
2	63	34	61	36	41	22	43
17	32	47	56	45	58	7	38
48	13	62	35	54	37	44	23
31	16	55	46	57	26	53	8
12	49	14	29	10	51	24	27
15	30	11	50	25	28	9	52
```

Example: Eight queens puzzle

- How to solve?
- Possible optimizations?
- Reusable knight?

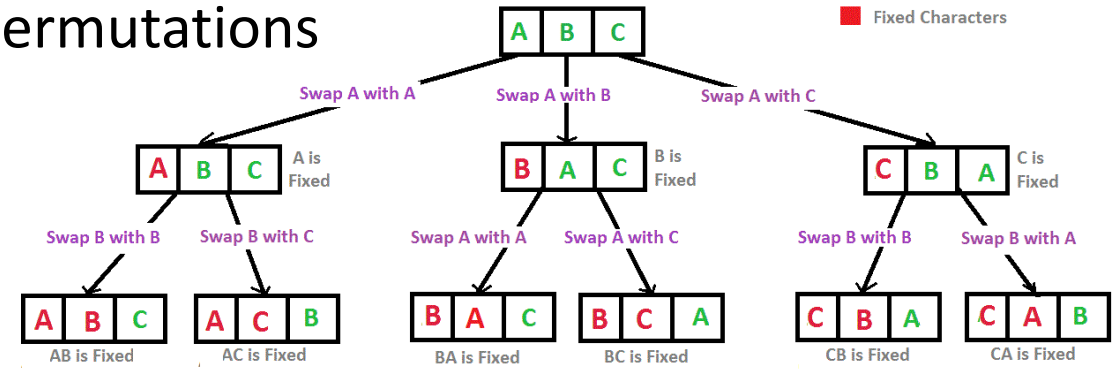


Example: generating permutations



Recursion Tree for Permutations of String "ABC"

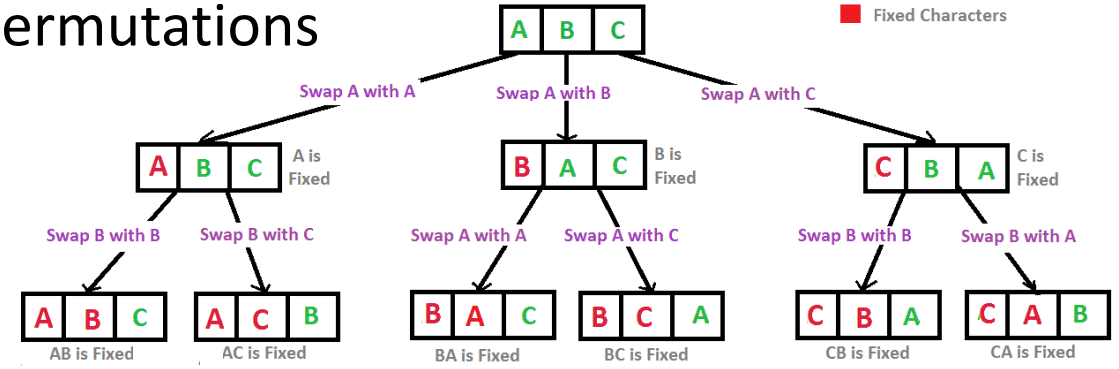
Example: generating permutations



Recursion Tree for Permutations of String "ABC"

```
def permute_string(str):
    permute_array(list(str))

def permute_array(a, swapping = 0):
    if swapping == len(a) - 1:
        print("".join(a))
        return
    for i in range(swapping, len(a)):
        a[swapping], a[i] = a[i], a[swapping]
        permute_array(a, swapping+1)
        a[i], a[swapping] = a[swapping], a[i]

permute_string("abc")
```

Example: generating permutations



Recursion Tree for Permutations of String "ABC"

```
def permute_string(str):
    permute_array(list(str))

def permute_array(a, swapping = 0):
    if swapping == len(a) - 1:
        print("".join(a))
        return
    for i in range(swapping, len(a)):
        a[swapping], a[i] = a[i], a[swapping]
        permute_array(a, swapping+1)
        a[i], a[swapping] = a[swapping], a[i]

permute_string("abc")
```

倒過來念的是小狗
倒過來念狗是小的
念小狗的倒是來過
念的是小狗倒過來
念的是小狗倒來過
小狗過來是倒念的
小狗念的是倒過來
小狗是倒過來念的
狗小是倒過來念的
是小狗的念倒過來

Time complexity?

Some other optional exercises (data structure, iteration, recursion)

How to revert (instead of revert print) a single linked list?

How to, starting from the tail, revert every k elements of a linked list?

e.g. k = 3, lst = 0, 1, 2, 3, 4, 5, 6, Output: 0, 3,2,1, 6,5,4

A two-dimensional matrix, increasing everywhere from left to right, and from bottom to top. Search if a number is in this matrix.

14	20	25	31	46
8	10	11	22	40
7	9	10	19	29
5	6	8	12	22
1	4	7	10	21

How to rewrite recursion into iteration?

What's next in algorithms?

- Sort (many algorithms)
- Search on graph
- Dynamic programming

Review & Possible content for the 1st Quiz:

Concepts and basic operations of array, linked list, dict, set, stack, queue

Concepts of binary tree, graph

Time complexity of access, insert, delete, search,

    append (amortized), pop (amortized) for array and linked list in Python

How to realize linked list, stack, queue, binary tree in Python or pseudo code

Recursion: concept,

    application to pre-order, in-order, post-order walk of binary tree

Recursion: other applications

Backup / more fun:

1. Try to use actual data – introduction to Taiwan traffic data.
2. Common pitfalls (坑s) in Python
3. Code exercise with the recursion algorithm

Common pitfalls (坑s) in Python

Spelling is important

为什么我写代码时总是手滑把main打成mian?

关注问题 | 写回答 | 邀请回答 | 好问题 55

查看全部 225 个回答

魔某人
游戏工程师，魔幻小说家

Albert Wang 等 4,333 人赞同了该回答

你的大脑:

```
#include <cstdio>
#include <thread>

void 左手() {
    putchar('a');
}

void 右手() {
    putchar('m');
    putchar('i');
    putchar('n');
}

int main() {
    std::thread 左手线程(左手);
    std::thread 右手线程(右手);
    左手线程.join();
    右手线程.join();

    return 0;
}
```

你的手:

```
mian
```

python3的print("床前明月光")打不出中文是怎么回事？？

补充说明： [图片] 这个真的和中文符号没关。中文字符问题很容易找，尤其是在代码量不大的初学时。显示全部

关注问题 | 写回答 | 邀请回答 | 好问题 5 | 5 条评论 | 分享 | 修改问

45 个回答 默认排序

大铀子
渣渣日语

2,469 人赞同了该回答

python3的print（“床前明月光”）打不出中文是怎么回事呢？python3的print（“床前明月光”）相信大家都很熟悉，但是python3的print（“床前明月光”）打不出中文是怎么回事呢，下面就让小编带大家一起了解吧。

python3的print（“床前明月光”）打不出中文，其实就是**用了中文标点**，大家可能会很惊讶python3的print（“床前明月光”）怎么会打不出中文呢？但事实就是这样，小编也感到非常惊讶。

这就是关于python3的print（“床前明月光”）打不出中文的事情了，大家有什么想法呢，欢迎在评论区告诉小编一起讨论哦！

编辑于 06-14

▲ 赞同 2469 ▼ | 139 条评论 | 分享 | 收藏 | 喜欢 | ...

Common pitfalls (坑s) in Python

0. Arrays count from 0. Guess what you get:

```
print(range(3))
```

range(3)

```
for i in range(3):
    print(i)
```

0, 1, 2

```
for i in range(0,3,1):
    print(i)
```

0, 1, 2

```
for i in range(3,0,-1):
    print(i)
```

3, 2, 1

Common pitfalls (坑s) in Python

1. Arrays are passed by reference, not value

```
a = [1,2,3]
b = a
print(id(a))
print(id(b))
```

you get the same object id

```
b[2] = 4
print(b)
print(a)
```

[1, 2, 4] and [1, 2, 4]

Common pitfalls (坑s) in Python

1. Arrays are passed by reference, not value

```
lst = ['a'] * 3      # [‘a’, ‘a’, ‘a’]

lst = [[]] * 3       # [[], [], []] 一時爽
lst[0].append(1)     # [[1], [1], [1]] /(ㄒoㄒ)/~~

Should use:

lst = [[] for i in range(3)]
```

Common pitfalls (坑s) in Python

2. Default argument of functions are initialized at definition.

```
def add_to_list(addition, lst=[]):
    lst.append(addition)
    return lst

a = add_to_list(1, [1, 2, 3])

b = add_to_list(1, [1, 2, 3])

c = add_to_list(1)

d = add_to_list(1)
```

Common pitfalls (坑s) in Python

2. Default argument of functions are initialized at definition.

~~def add_to_list(addition, lst=[]):~~
~~   lst.append(addition)~~
~~   return lst~~

```
def add_to_list(addition, lst=None):
    if lst is None:
        lst = []
    lst.append(addition)
    return lst
```

Common pitfalls (坑s) in Python

3. Be careful about variable types.

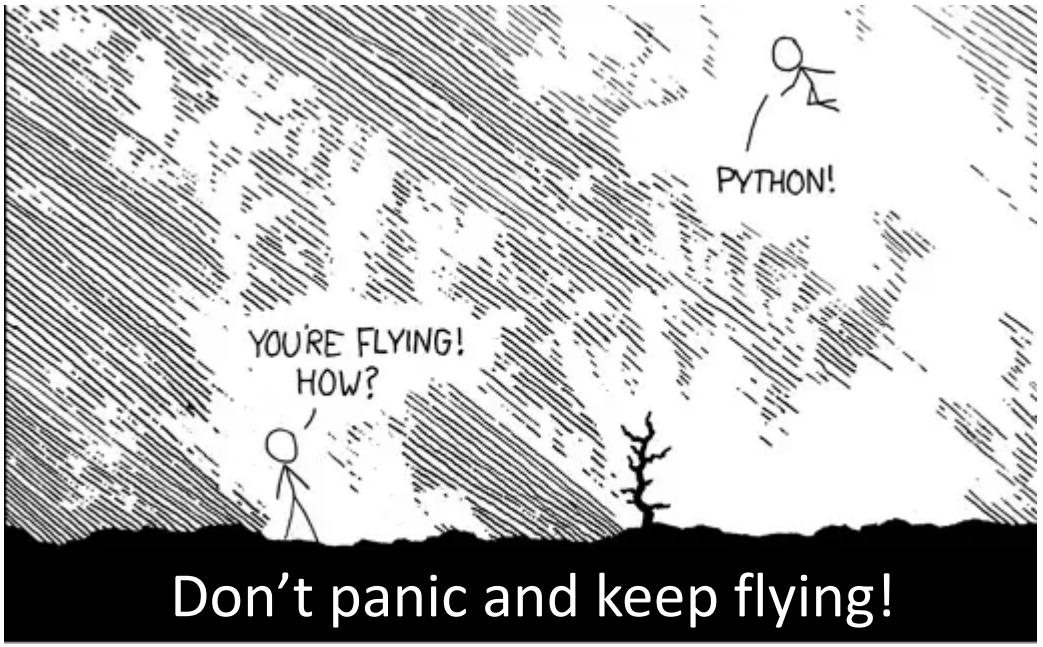
Duck type is very convenient, but also dangerous.

```
num_death = 2
print("全省{}人死亡".format(num_death))

# Your 坑 colleague might pass you:
num_death = ""
print("全省{}人死亡".format(num_death))
```

Common pitfalls (坑s) in Python

4. Use logical operations for logical variables

```
print(True + True) # 2. Should: True and True
```



Don’t panic and keep flying!