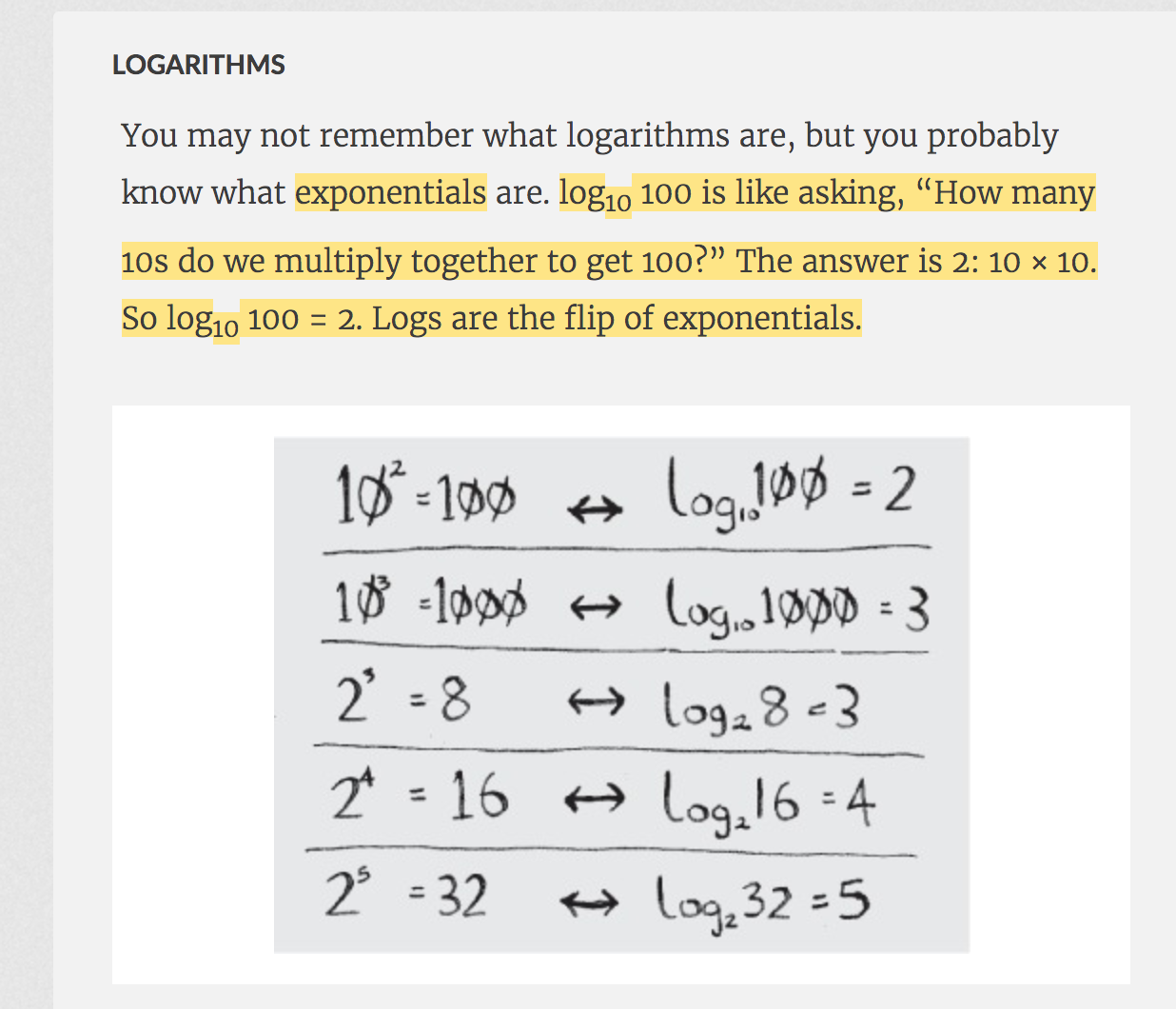
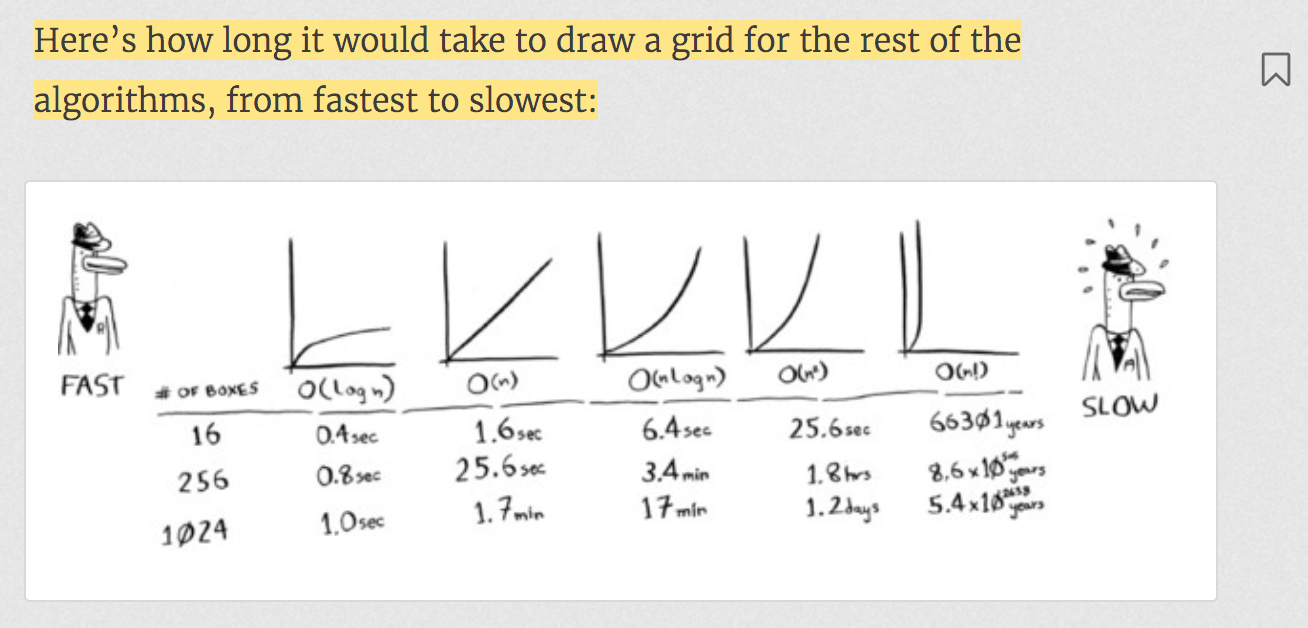
# Logarithm



# Big O Notation

Big O doesn’t tell you the speed in seconds. Big O notation lets you compare the number of operations.

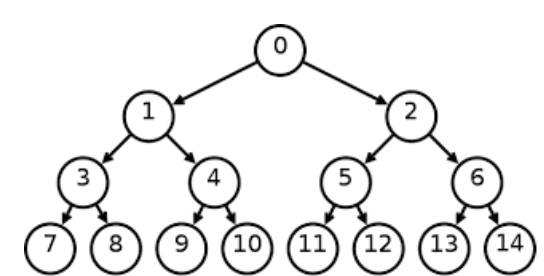
# 



Learn Logarithm

<https://www.youtube.com/watch?v=ZIwmZ9m0byI>

How do you decide the base of log?



Height of this tree is 3 that is log2 14.

How do you know the base is 2?

Log2 14 = x

3.81=x

It’s a binary tree so if you use log with base 2, it gives correct height of the tree.

If it’s ternary tree, you need to use base 3 and so on to get the height of the tree.

What is Log2 8?

X = log2 8

2^x = 8 (**exponential** form of log)

x = 3

what if it is hard to find the value of x

e.g. x = log2 11

2^x = 11

it is hard to find the value of x

So, log2 11 = log 11/log 2 (default base is 10) = 3.33

**Natural log (ln)**

ln 8 is same as loge 8

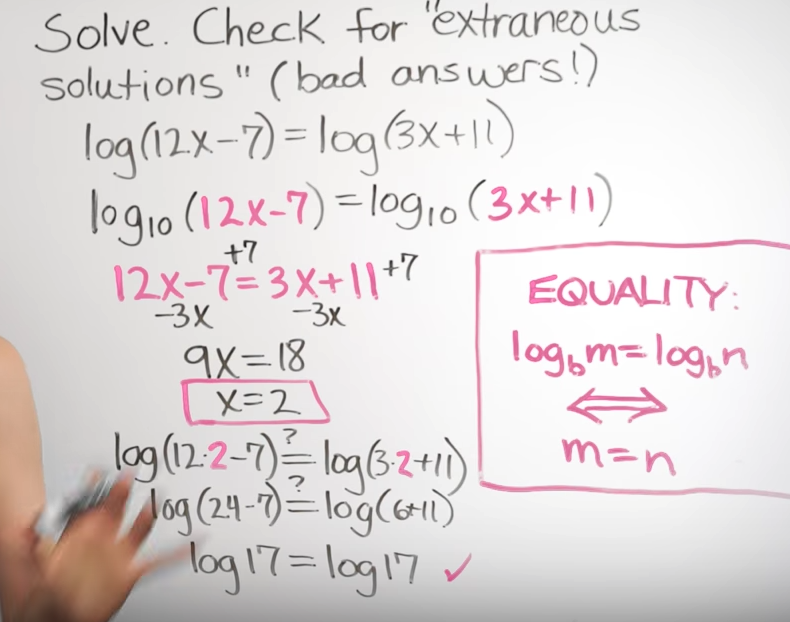
x= loge 8

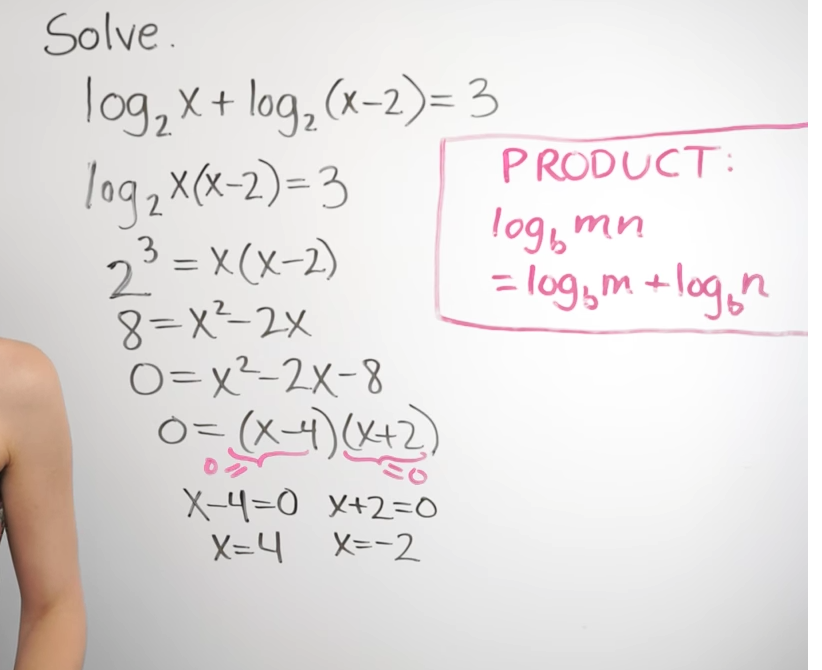
e^x = 8

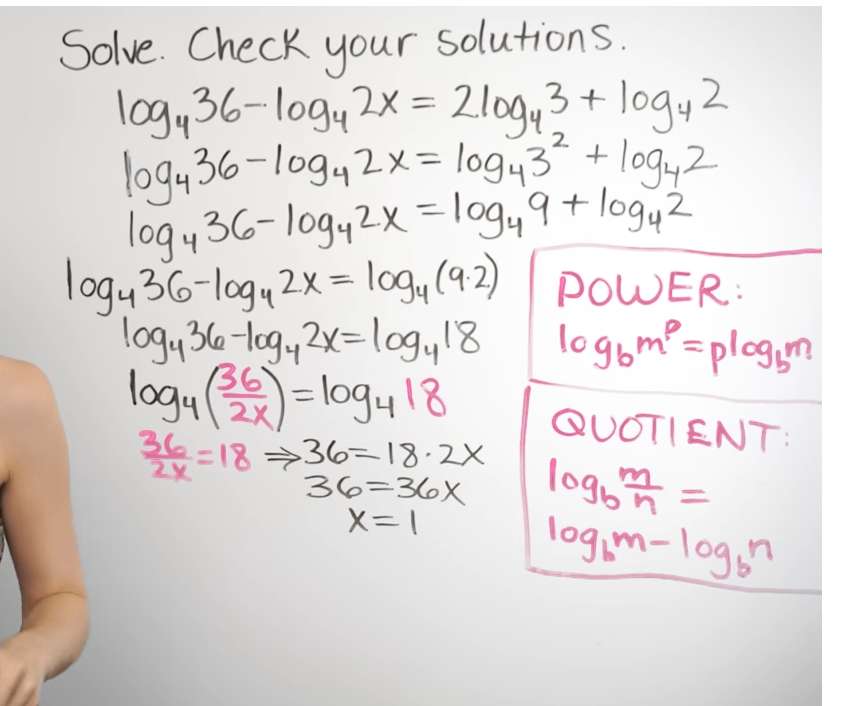
e^x = 2^3

x = 3

**Equality**







Remember : negative number is not considered in the world of logarithm.

e.g. log -8 is undefined.

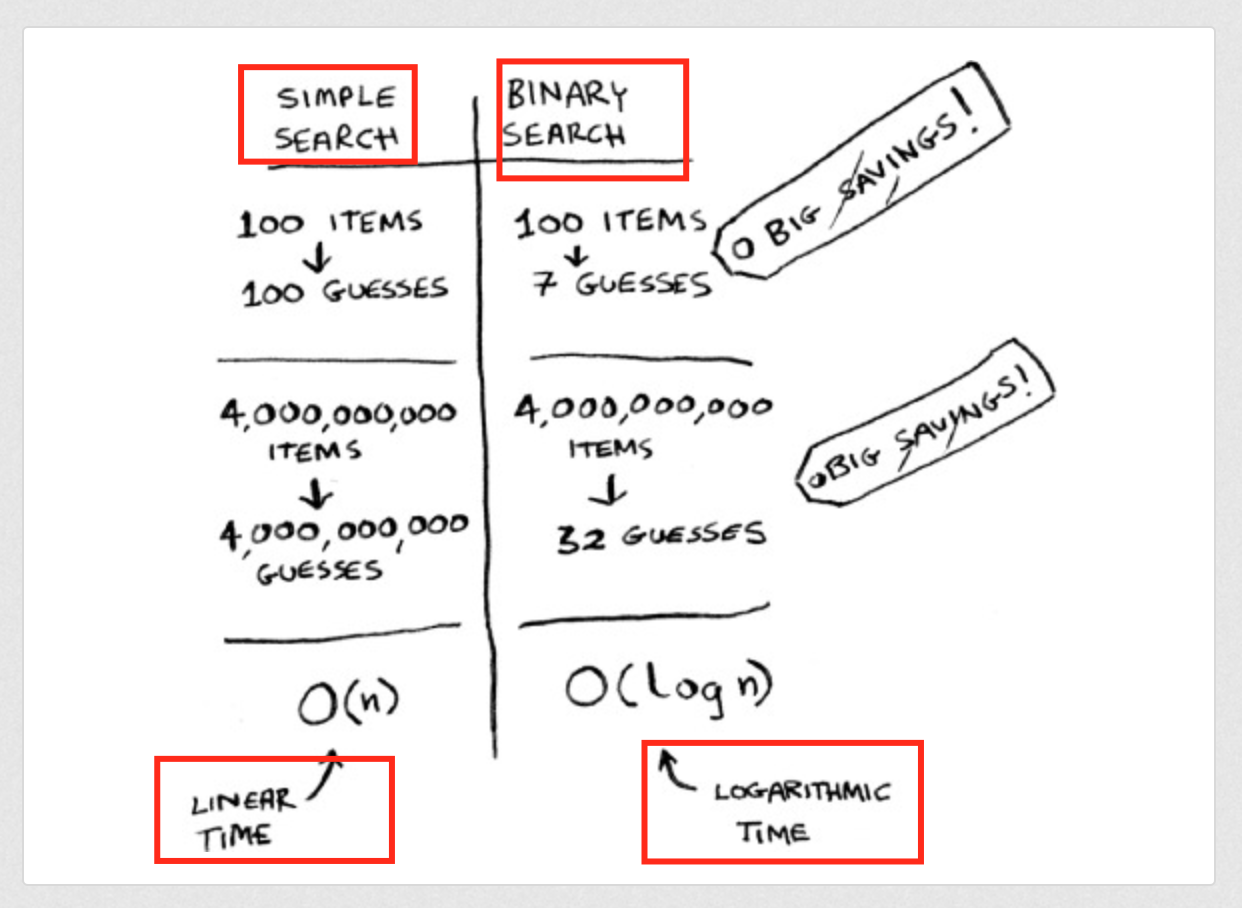
Logarithm real life example

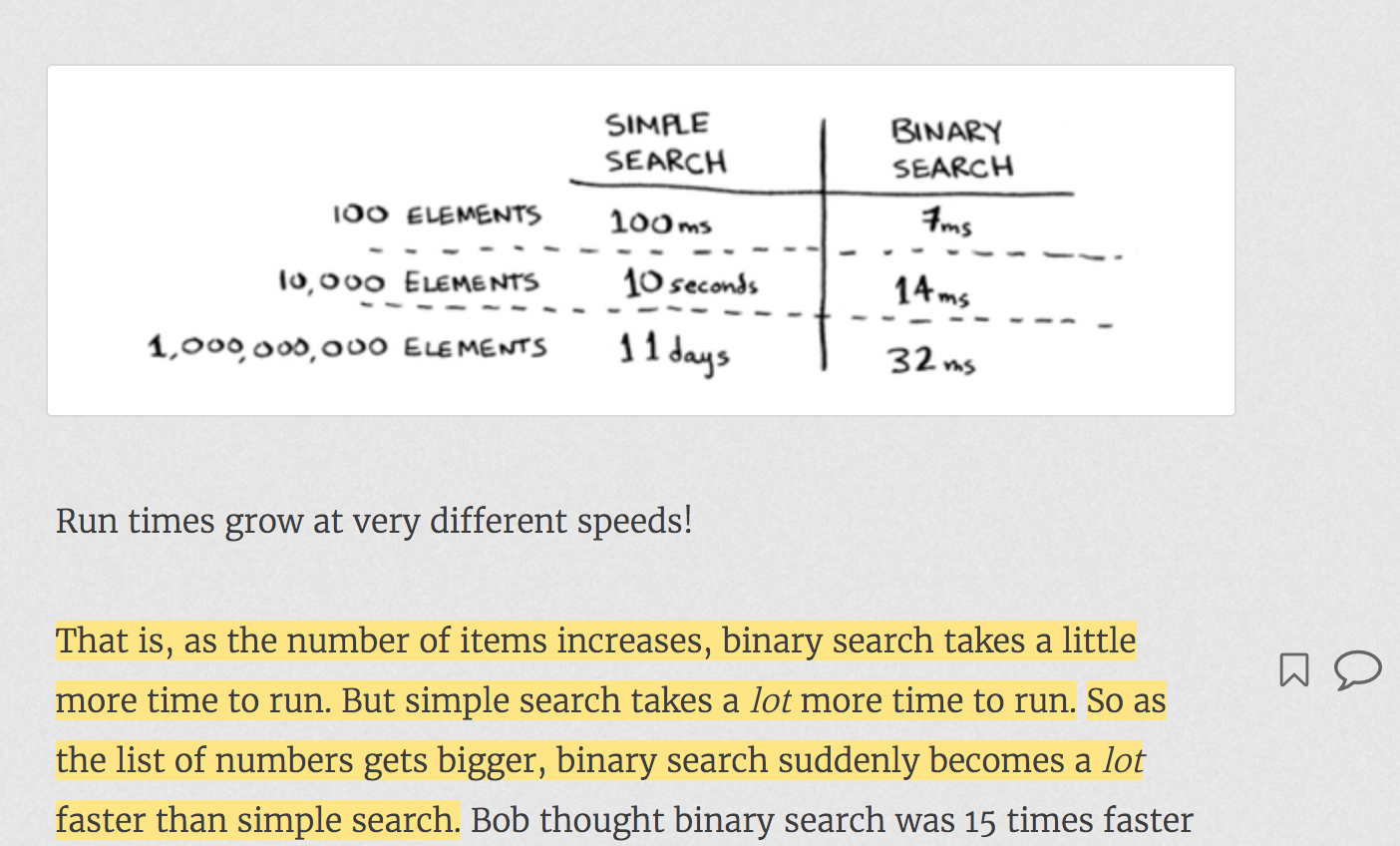
<https://www.youtube.com/watch?v=1dUSNdZspQc>

# Binary Search

* Binary Search works only on Sorted Array.
* It doesn’t work on linked list because it has to access element by index.
* It takes O(log n).

*o  
 /\  
 o o  
 /\ /\  
 o o o o  
 /\  
o o*In binary search, to search an element, you divide an array into two and then you keep searching on its one side.  
This is like binary search tree. It takes max O(log n) to search an element in bst.





## When you need to work with array:

binarySearchAlgorithmRecursive(array, 0, array.**length** - 1, elementToSearch)

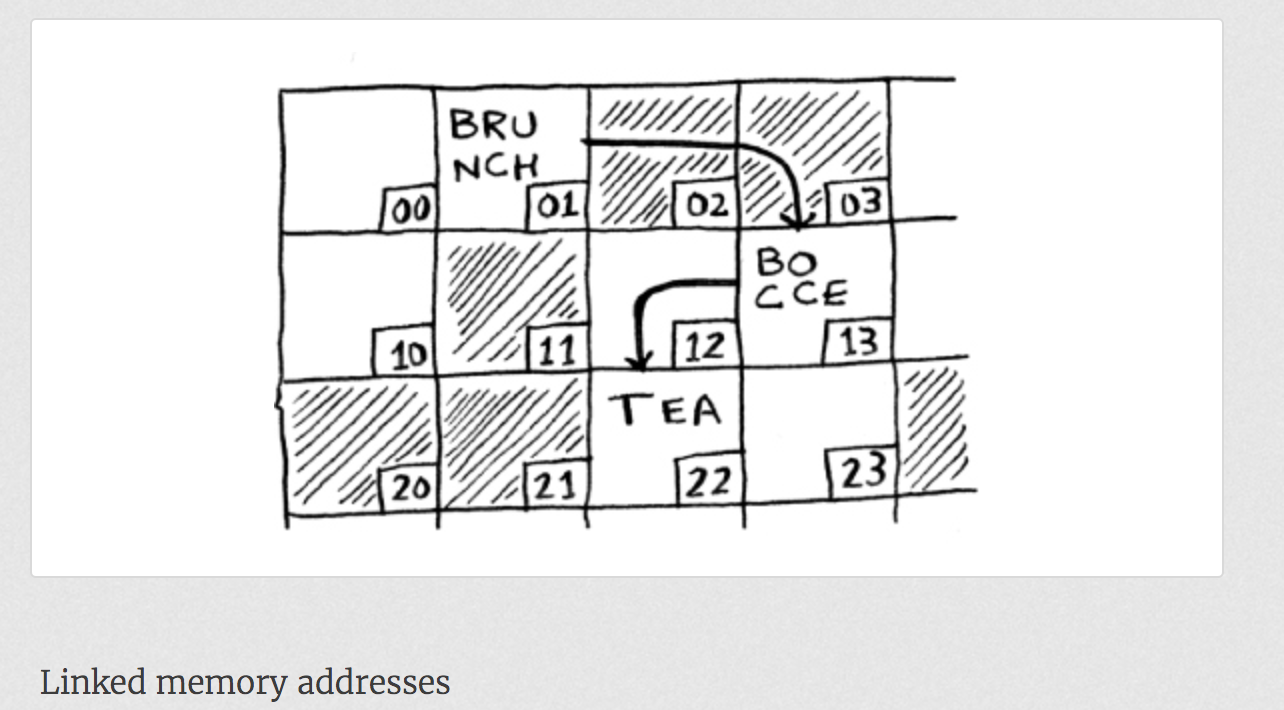
* Always pass start and end element position in array to recursive method.
* One of the Exit condition will be if(start<end)…
* When you need to convert recursive method into iterative method, extra passed parameters to recursive method becomes local variables and after that that you need to add a while loop for reoccurring code.

# Arrays and List

# 

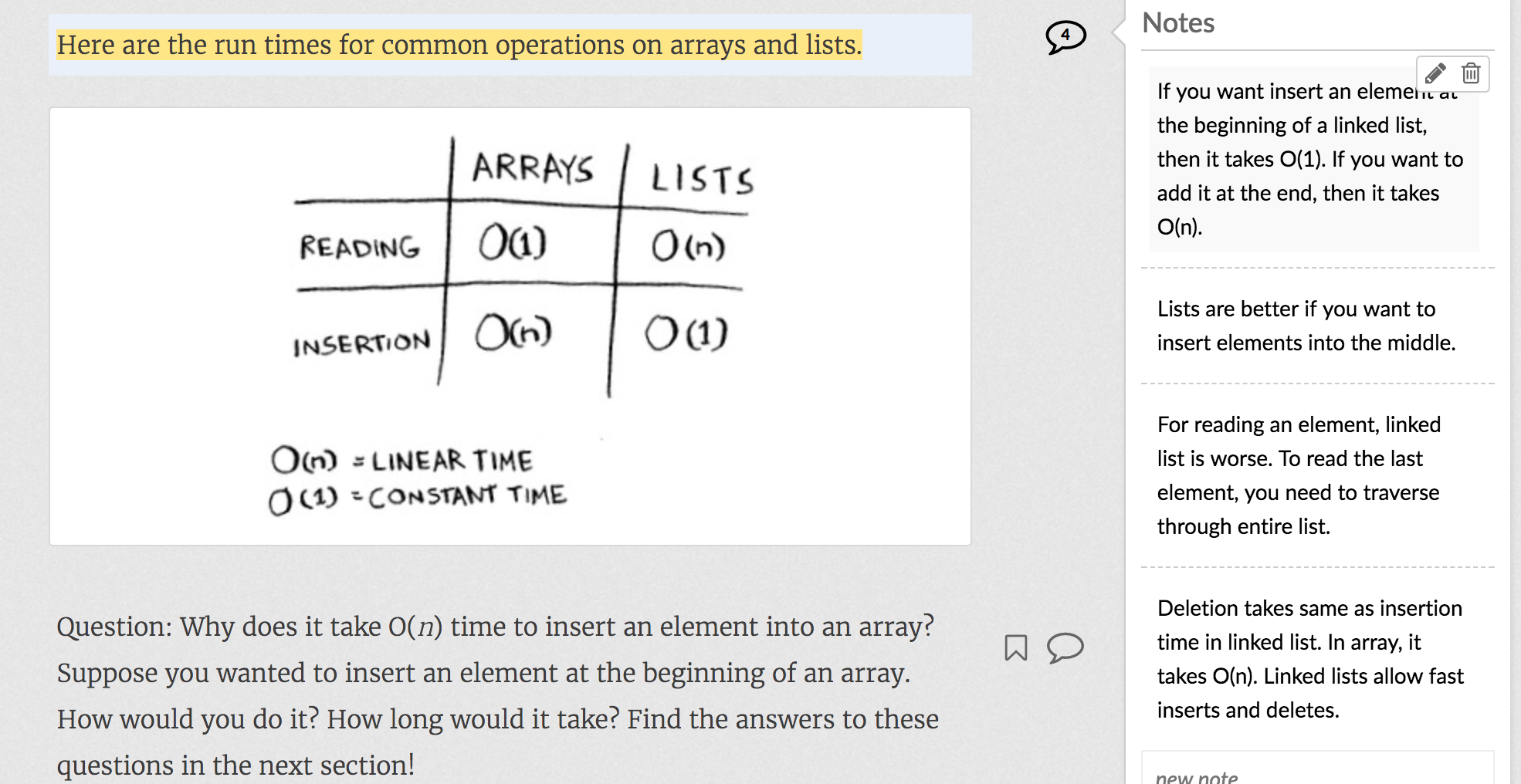
 If you’re out of space and need to move to a new spot in memory every time, adding a new item will be really slow.

With linked lists, your items can be anywhere in memory.



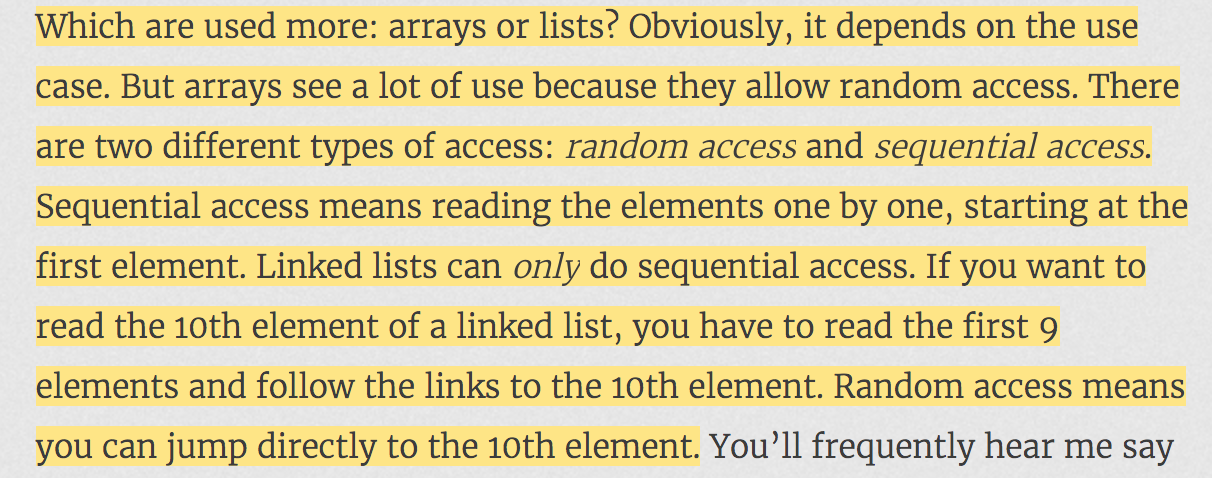
Adding an item to a linked list is easy: you stick it anywhere in memory and store the address with the previous item.

Suppose you want to read the last item in a linked list. You can’t just read it, because you don’t know what address it’s at. Instead, you have to go to item #1 to get the address foritem #2. Then you have to go to item #2 to get the address for item #3.



Lists are better if you want to insert elements into the middle.

# 

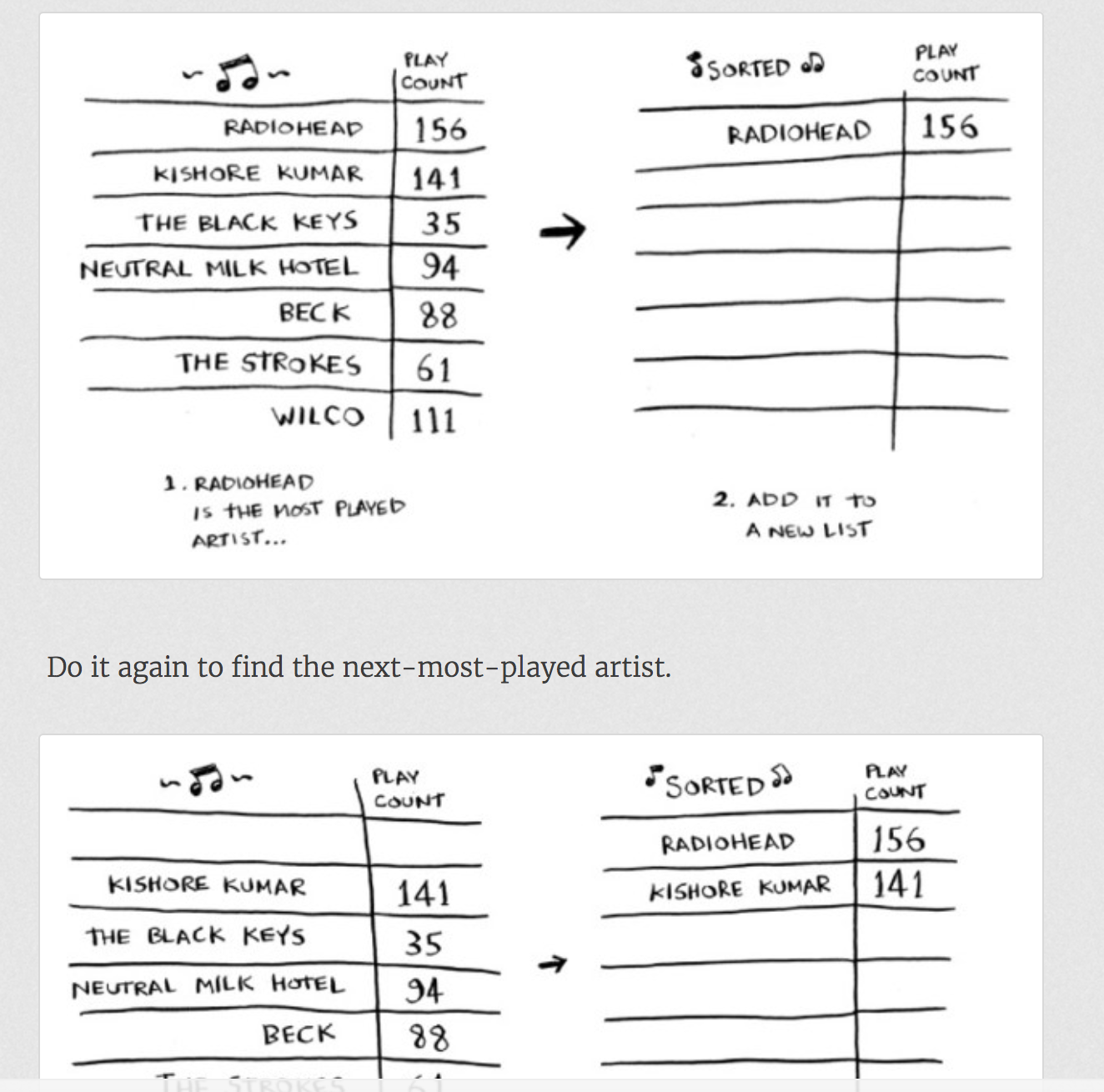


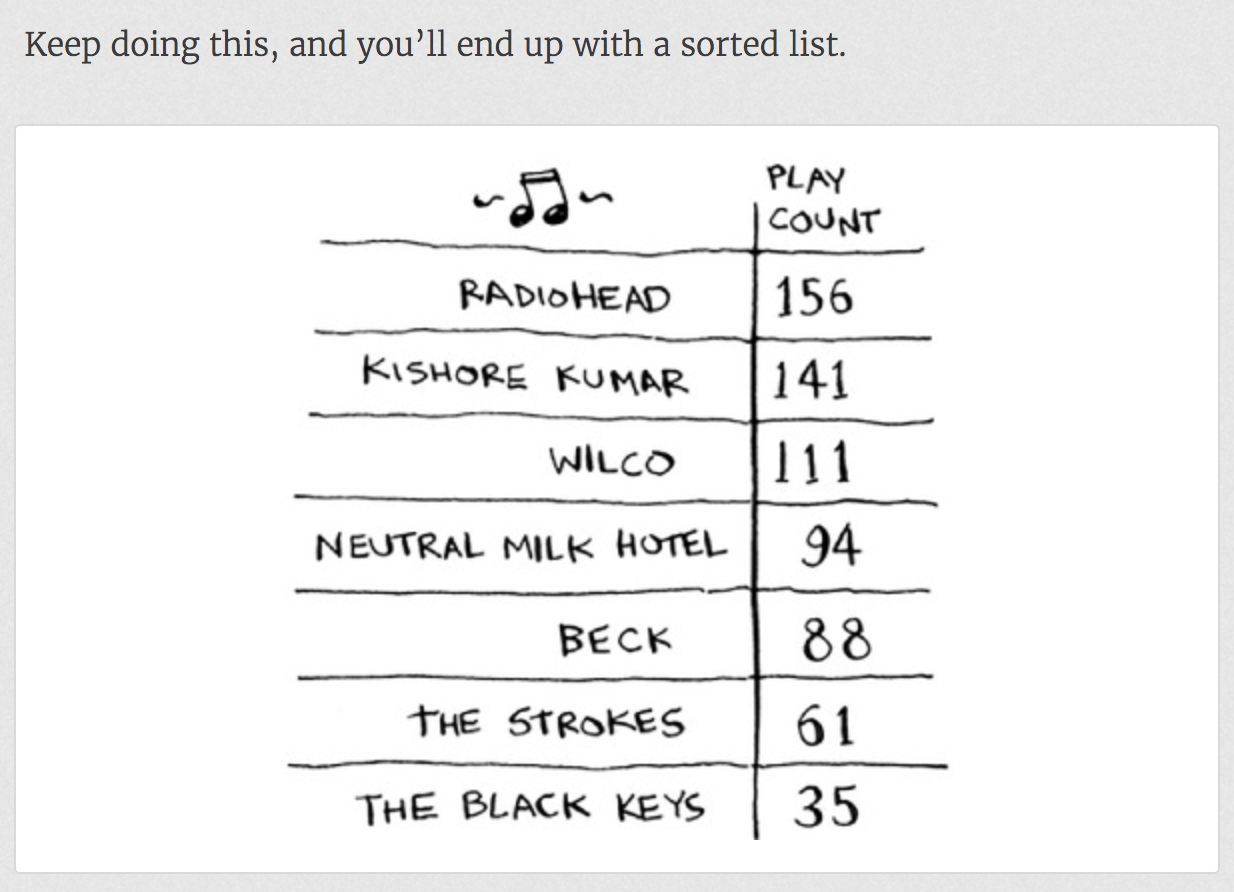
# Bubble Sort and Selection Sort

Selection sort is a stepping stone to quicksort, which I’ll cover in the next chapter. Quicksort is an important algorithm, and it will be easier to understand if you know one sorting algorithm already.

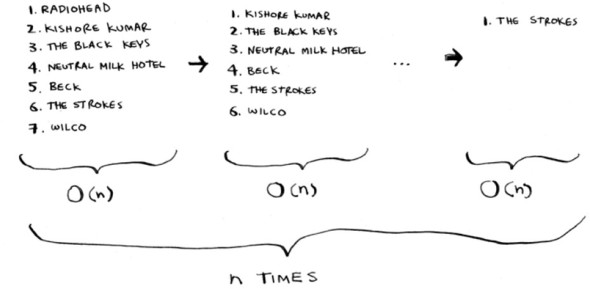
Bubble Sort:

You need to compare each element of an array with all other elements.





To find the artist with the highest play count, you have to check each item in the list. This takes O(*n*) time, as you just saw. So you have an operation that takes O(*n*) time, and you have to do that *n* times:



This takes O(*n* × *n*) time or O(*n*2) time.

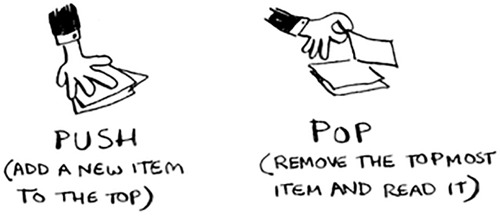
Bubble Sort compares first element with other elements and if first element > other element, then exchanges(swaps) them.

**Unlike to Bubble Sort, Selection Sort does not swap the elements after every comparison.**  
It **keeps min=i**. Compares each element of array with a[min]. if a[min]>a[other index], then min=other index.  
At the end, it swaps elements of a[i] and a[other index].

But both Bubble Sort and Selection Sort takes O(n2) operations.

# Recursion

The stack of sticky notes is much simpler. When you insert an item, it gets added to the top of the list. When you read an item, you only read the topmost item, and it’s taken off the list. So your todo list has only two actions: *push* (insert) and *pop* (remove and read).

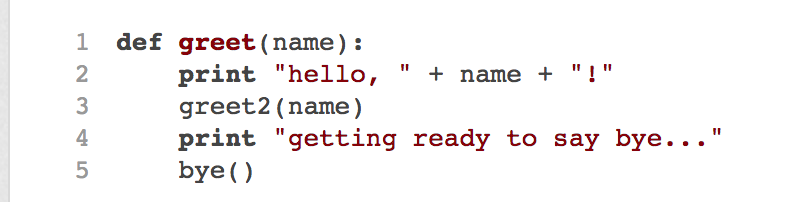


Let’s see the todo list in action.

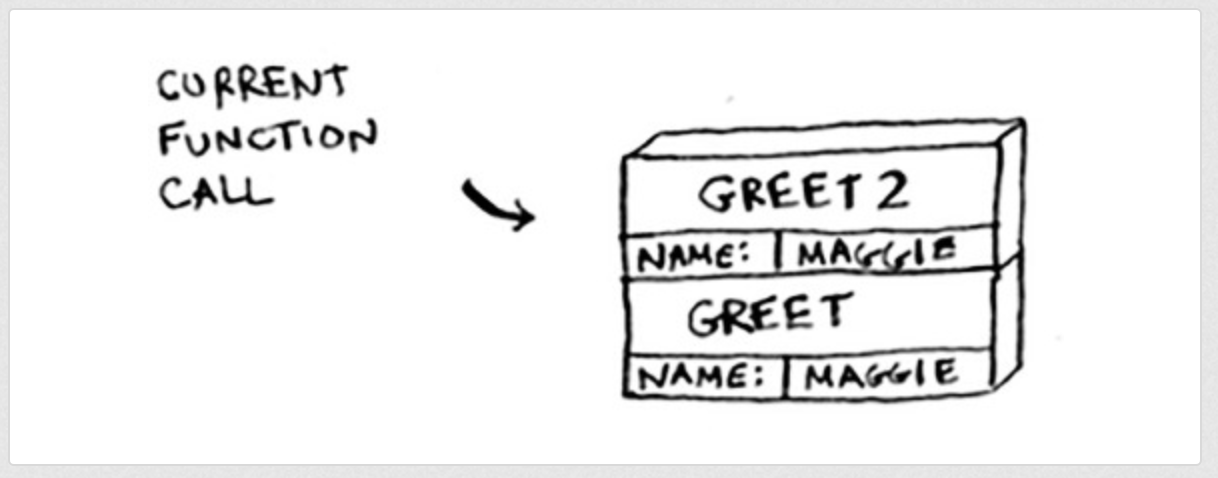


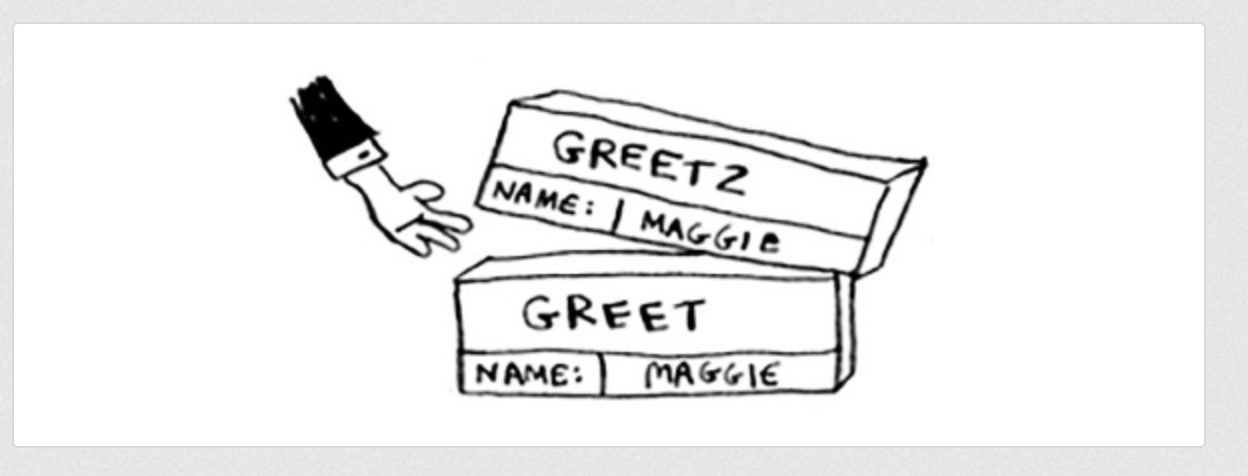
This data structure is called a *stack*.

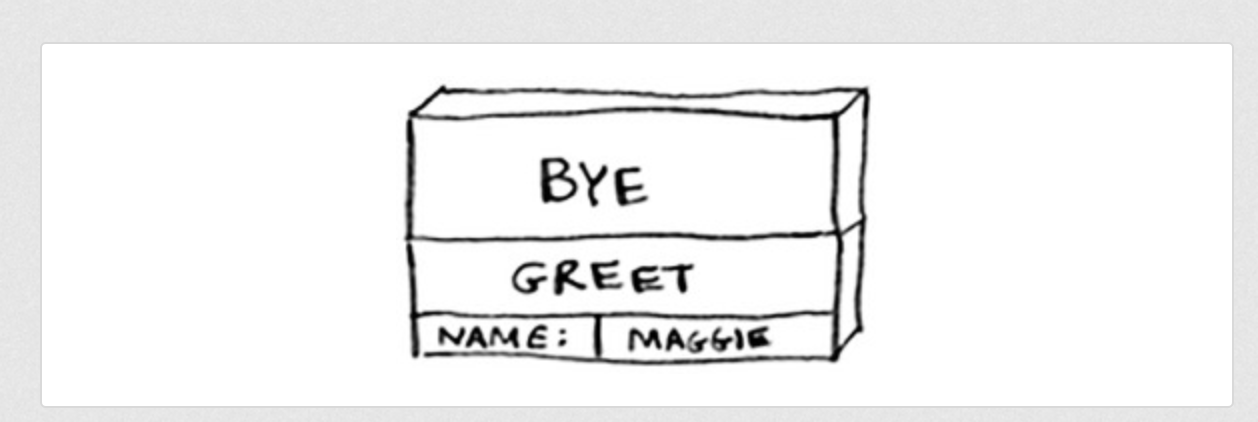
e.g.

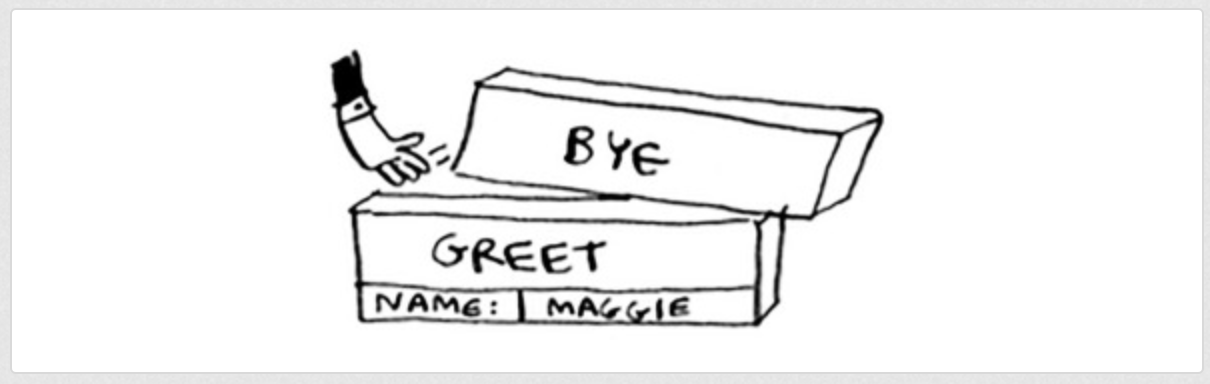












Converting Recursive function to Iterative function.

Recursive function call:

fact(n) {

if(n==0) return 0;

if(n==1) return 1;

return n \* fact(n-1)

}

|  |  |  |  |
| --- | --- | --- | --- |
| **step** | **Push to Stack** | **step** | **Pop from Stack** |
| 4 | If(x==1) return 1 |  |  |
| 3 | 2 \* fact(1) | 5 | 2 \* 1 = 2 |
| 2 | 3 \* fact(2) | 6 | 3 \* 2 = 6 |
| 1 | fact(3) | 7 | 6 |

Recursive function can be converted to iterative approach as follows:

fact(n) {

int total=n; --- from n \* fact(n-1) sentence, total=n

while(n>1) { ---- from exit condition if(n==0) and if(n==1)

total = total \* (n-1); ---- from n\*fact(n-1), total = total \* (n-1)

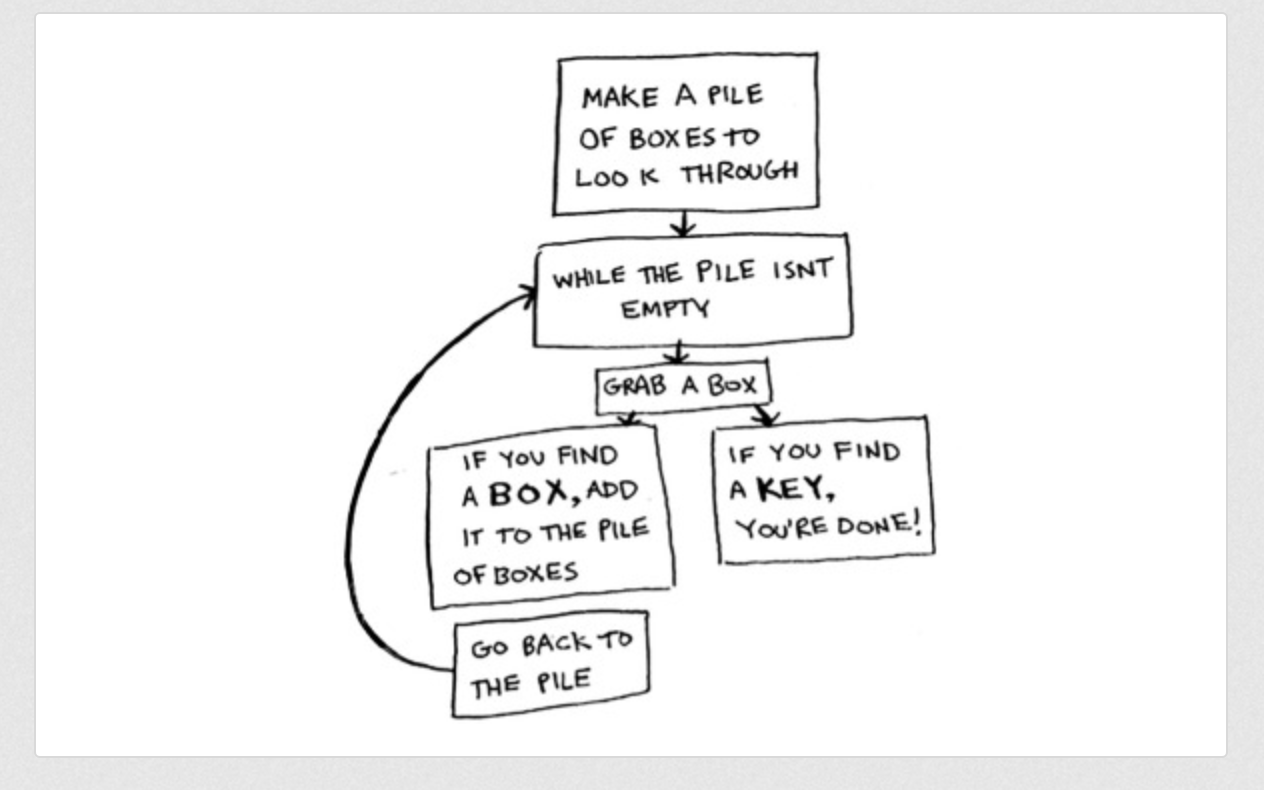
n = n-1; ----- from fact(n-1), n=n-1

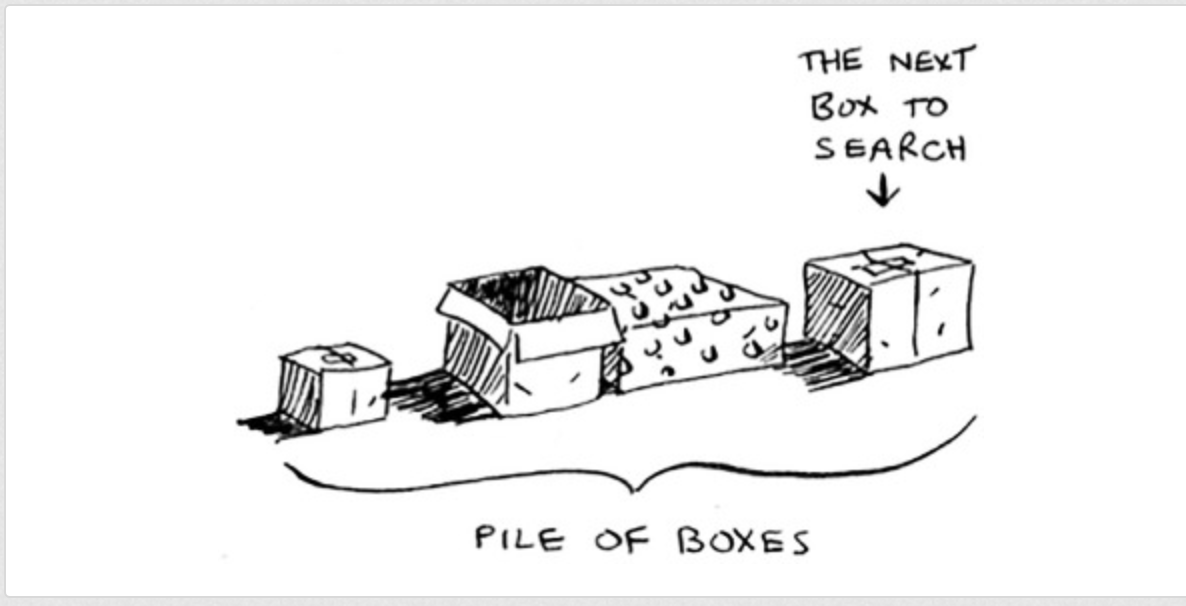
}

return total;

}

Iterative approach





Recursive approach

