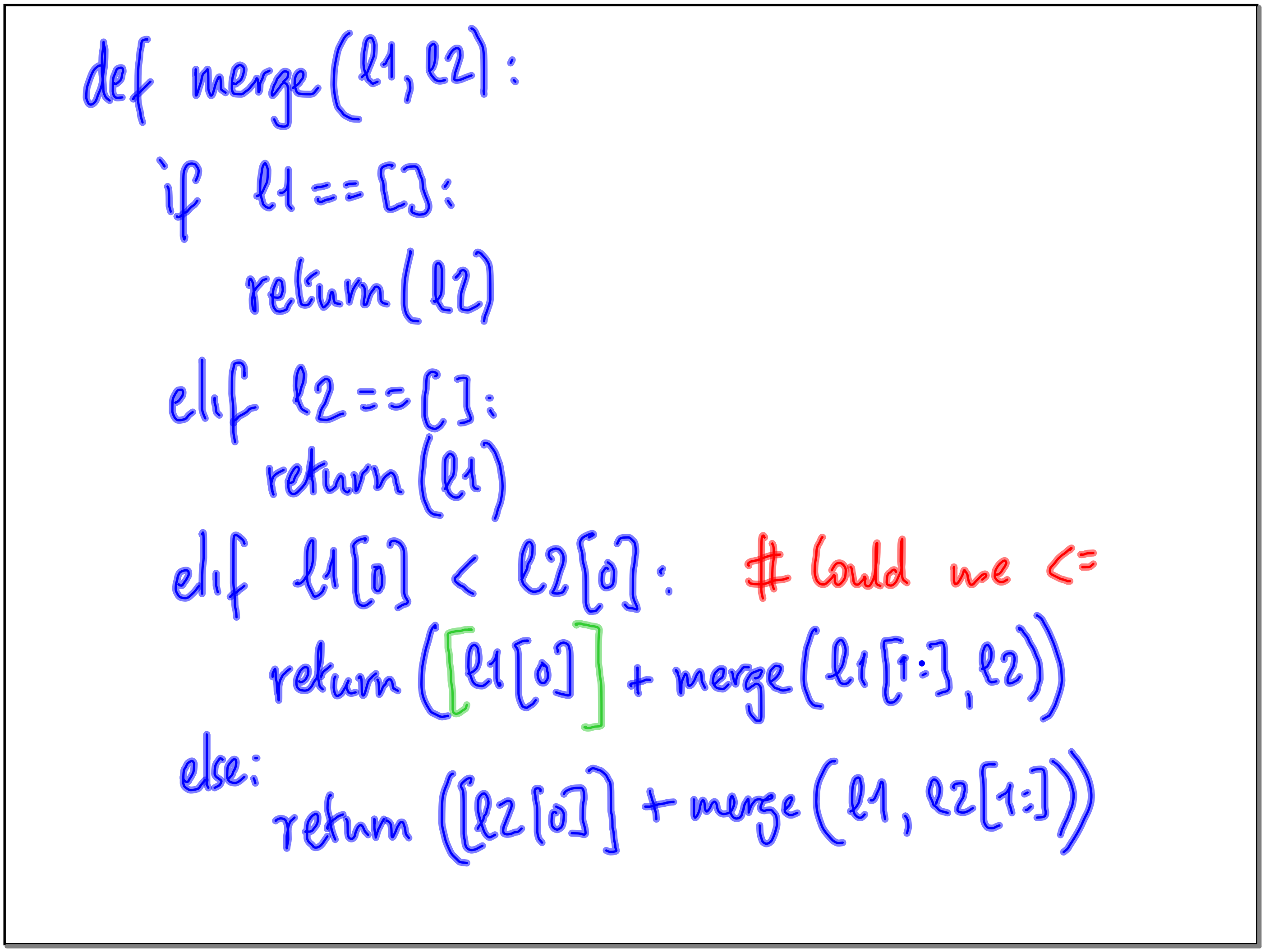


```
def merge(l1,l2):
    if l1==[]:
        return(l2)
    elif l2==[]:
        return(l1)
    elif l1[0] < l2[0]:   # could we <=
        return([l1[0]] + merge(l1[1:],l2))
    else:
        return([l2[0]] + merge(l1,l2[1:]))
```

Mergesort DIVIDE & CONQUER

What if we divide more intelligently?

Suppose we can identify median value in l

Split as

< median $1/2$ → sort l_1

+

median [median]

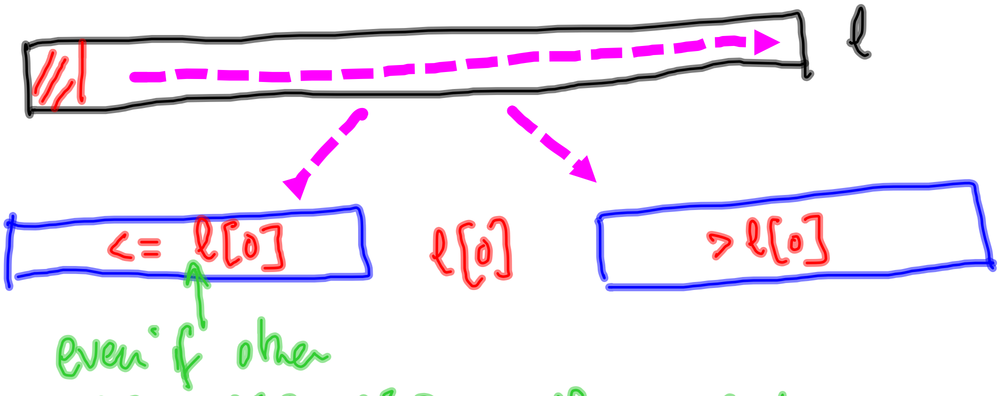
+

> median $1/2$ → sort l_2

Identifying median is "equivalent" to sorting!

Instead pick a "random" value to split the list

Always pick l[0] to split.



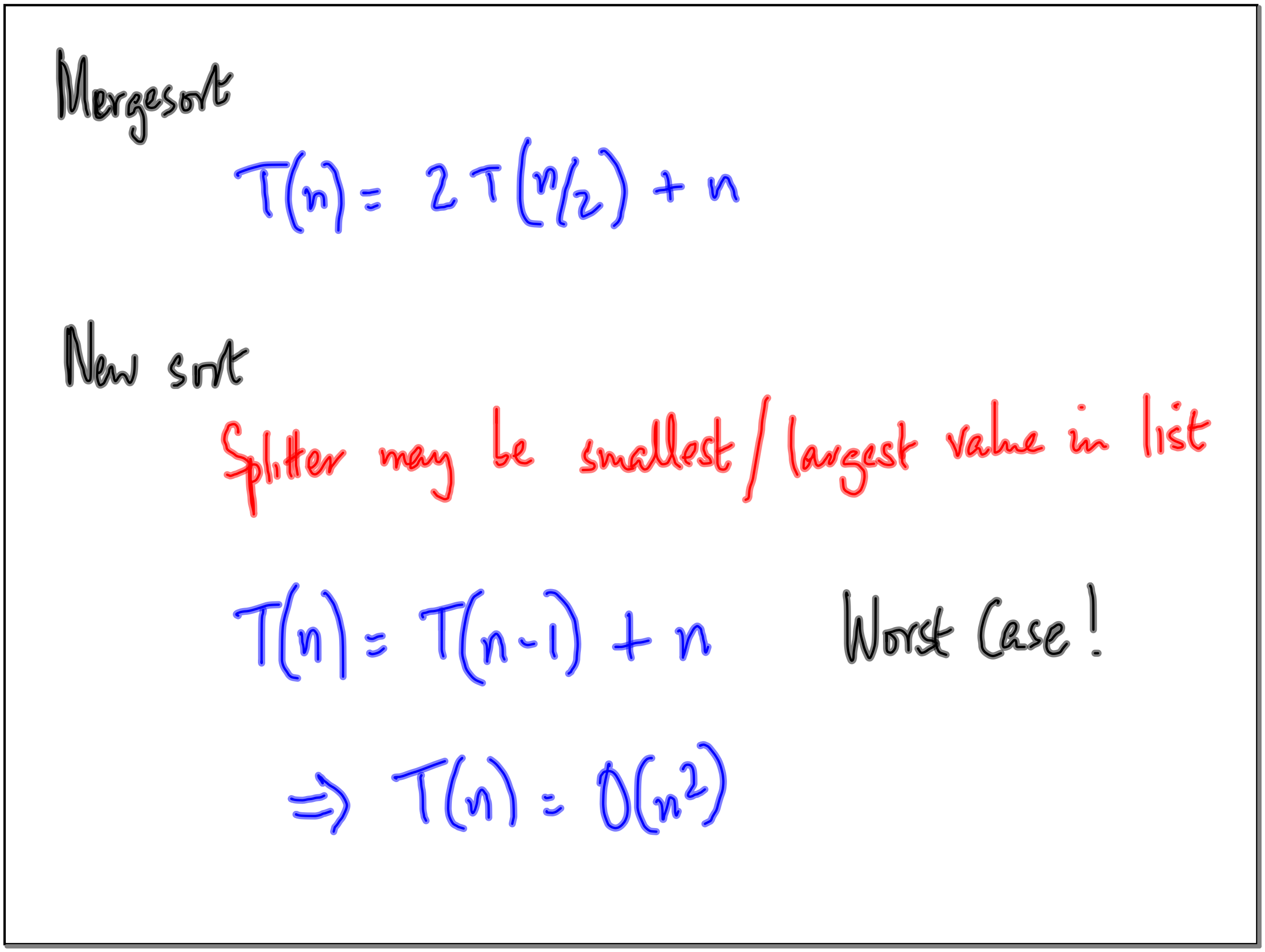
even if other l[i] == l[0], l[0] itself is not here

[<= splitter]   [splitter]   [>= splitter]

↓ sort   ↓ sort

lower + [splitter] + upper

Divides by value, not by position



Mergesort

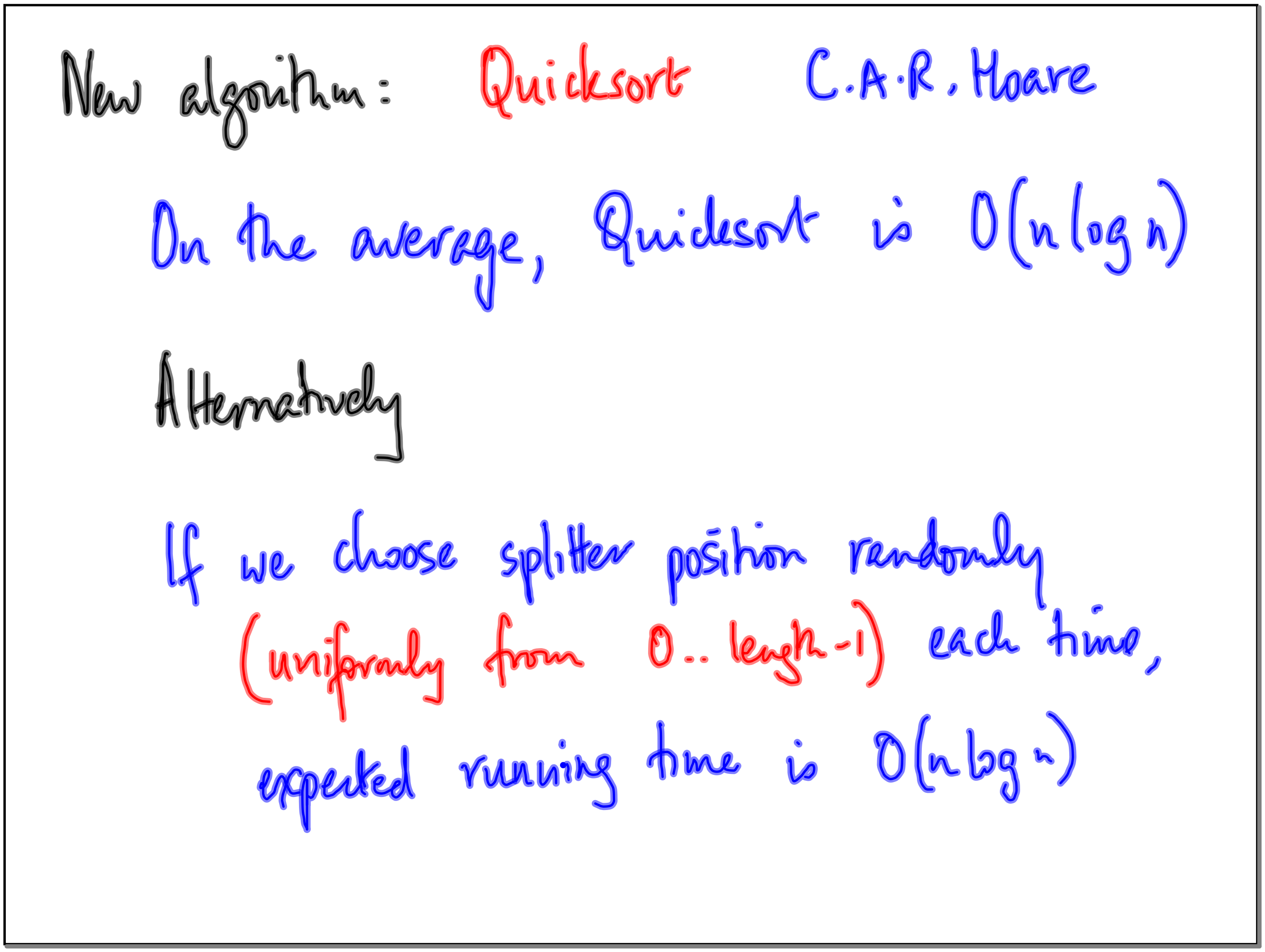
$$T(n) = 2T(n/2) + n$$

New sort

Splitter may be smallest / largest value in list

$$T(n) = T(n-1) + n$$ Worst Case!

$$\Rightarrow T(n) = O(n^2)$$



New algorithm: Quicksort C.A.R. Hoare

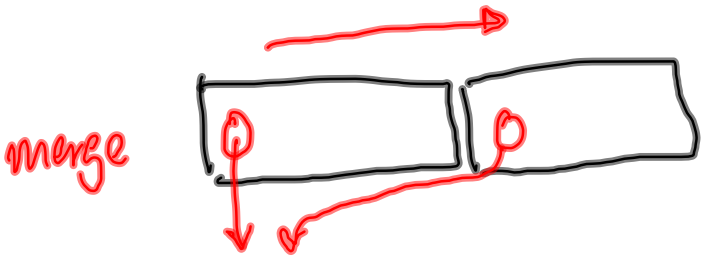
On the average, Quicksort is $O(n \log n)$

Alternatively

If we choose splitter position randomly (uniformly from 0..length-1) each time, expected running time is $O(n \log n)$

Space Complexity

Mergesort require separate list to merge



Quicksort

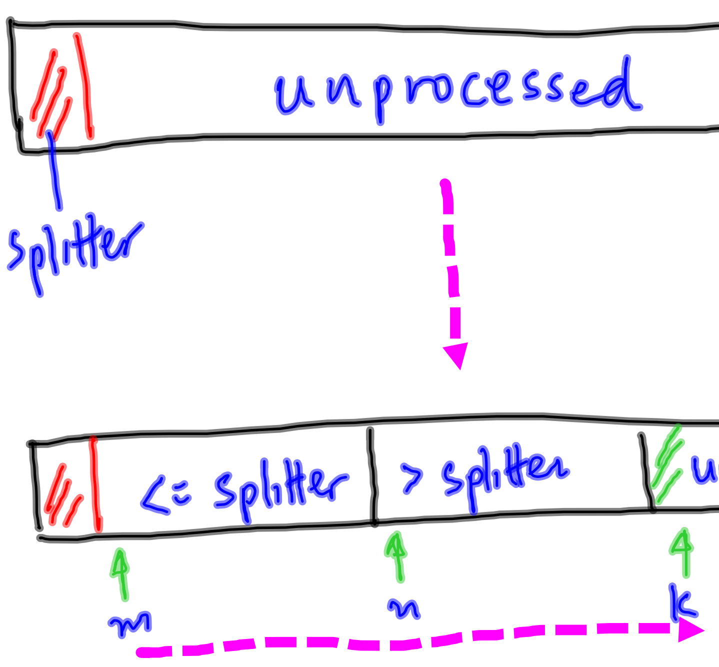
Inductive implementation has same problem

Space to create $\boxed{\leq \text{ splitter}}$

$\boxed{> \text{ splitter}}$

Can split more intelligently

Splitting a list in place



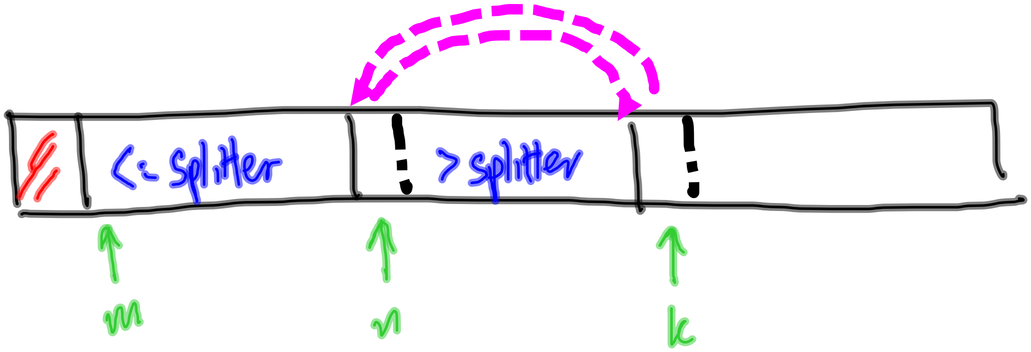
l[k] > splitter, increment k

l[k] <= splitter?

Expensive! Move > splitter block right?

Observation:

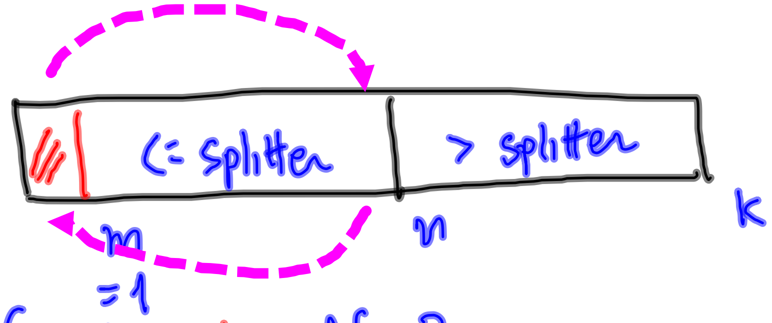
Need not preserve original list order in <= splitter and > splitter



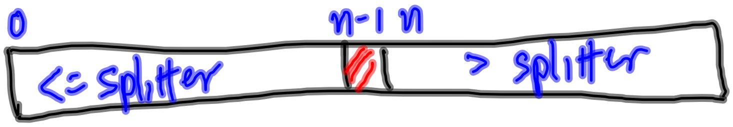
l[k] <= splitter   Exchange l[n], l[k]

```
(l[n], l[k]) = (l[k], l[n])   # swap
n = n+1                       # proceed
k = k+1
```

At the end:



Swap splitter (l[0]) with l[n-1]

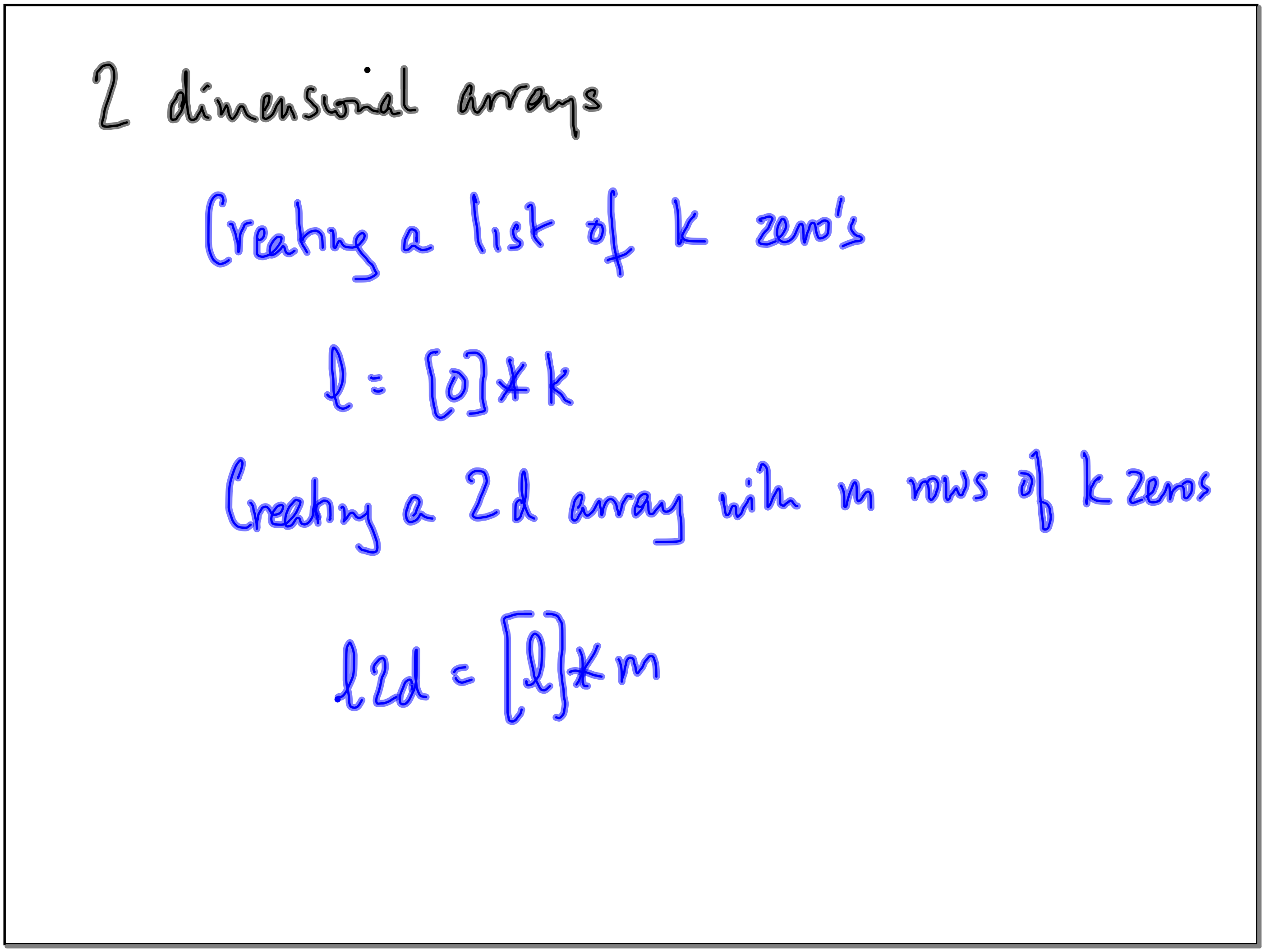


Modify quicksort to sort a segment l[left:right]

Initially sort l[0:len(l)]

sort l[0:n-1] sort l[n:]

Instructive exercise : Code this version of quicksort in Python



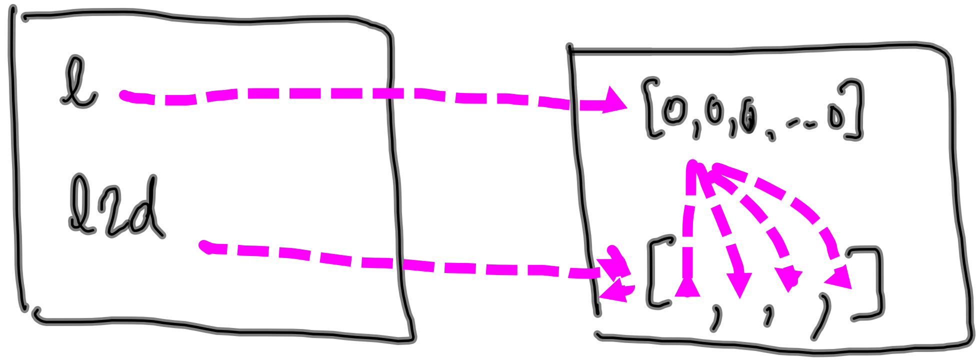
2 dimensional arrays

Creating a list of k zero's

```
l = [0]*k
```

Creating a 2d array with m rows of k zeros

```
l2d = [l]*m
```


l
l2d
[0,0,0,...0]
[, , ,]