Defining functions inductively

Numbers — factorial, fibonacci, gcd

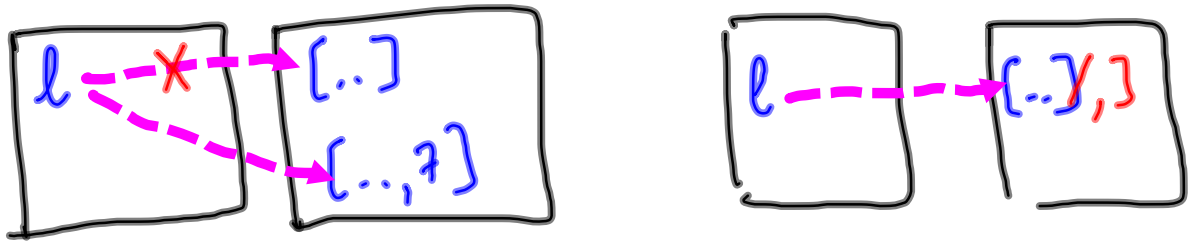
Structures like lists — Base case [] — length
Inductive step — sum

Before we proceed ...

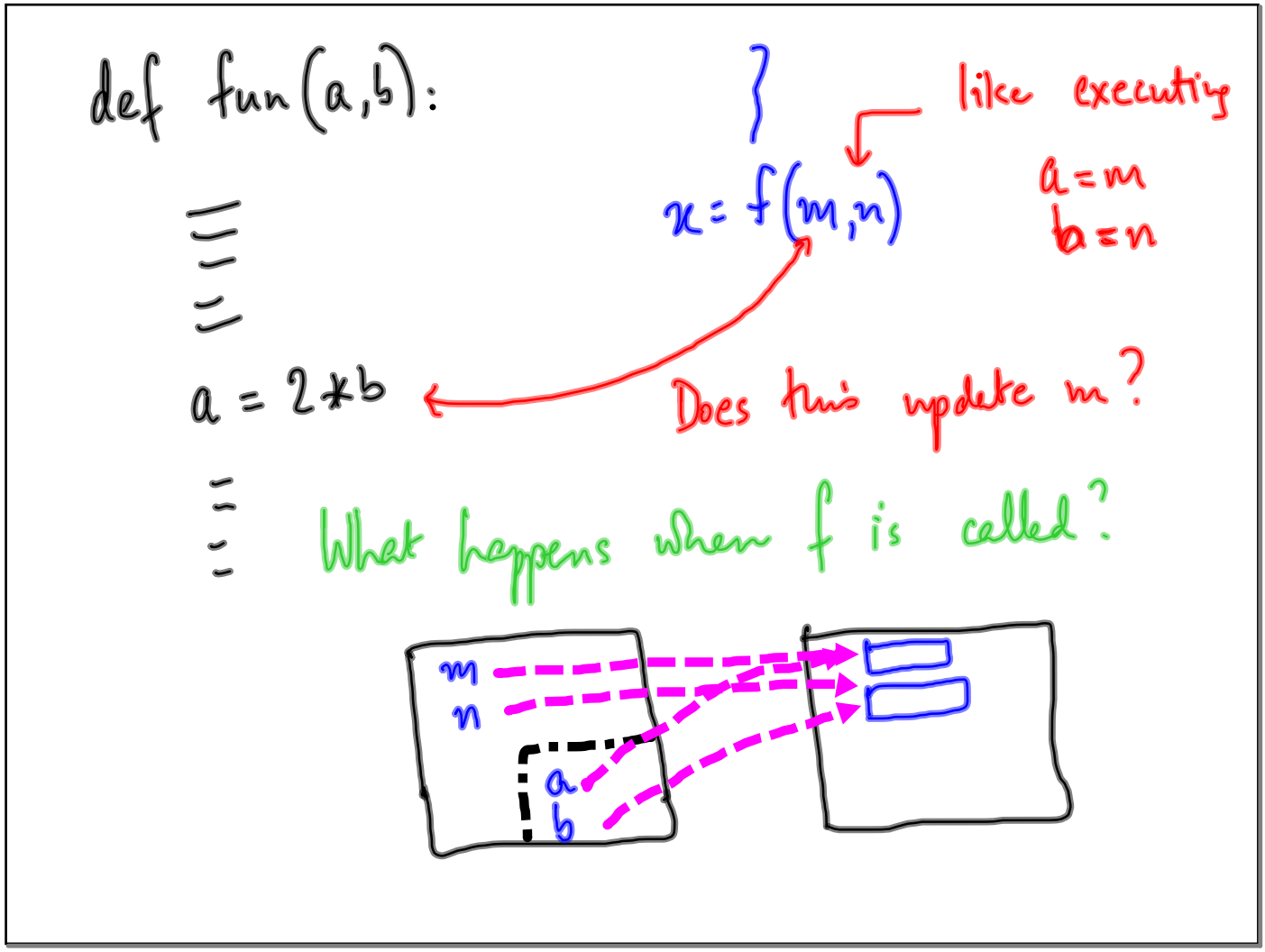
`l = l + [7]` vs `l.append(7)`

creates a new object

modifies existing object



Why is this important?


def fun(a,b):
like executing
x = f(m,n)
a = m
b = n
a = 2*b
Does this update m?
What happens when f is called?
m
n
a
b

```
def fun(a,b):
    .
    :
    a = a + [7]
    b.append(7)
```

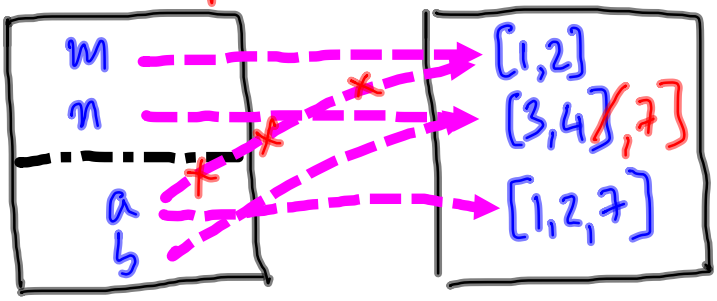
Creates new object. No effect on m

Modifies b in place, n is updated as well

m = [1,2]

n = [3,4]

x = f(m,n)



Sorting a list

Ascending

Built in function l.sort()

How does it work?

I give you a shuffled pack of playing cards

13 cards x 4 suits

[A, K, Q, J, 10, ..., 2] [S, H, D, C]

> >

Smallest? (C,2) < (C,3) < ... (C,A) < (D,2) ...

How would you sort it?

Algorithm 1

Separate the suits

Look for (C,2), then (C,3) ...

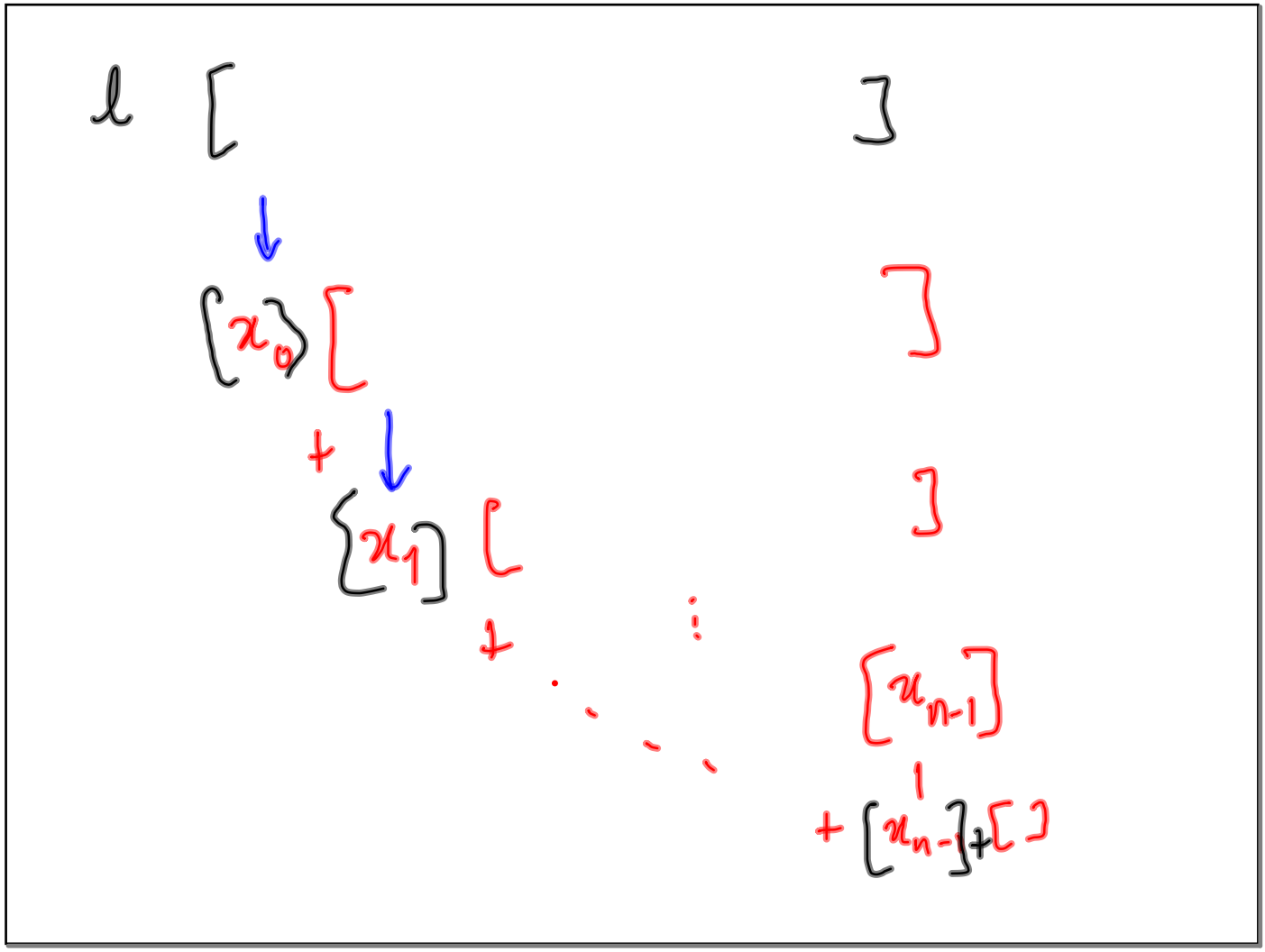
Selection Sort

Select smallest value at each step

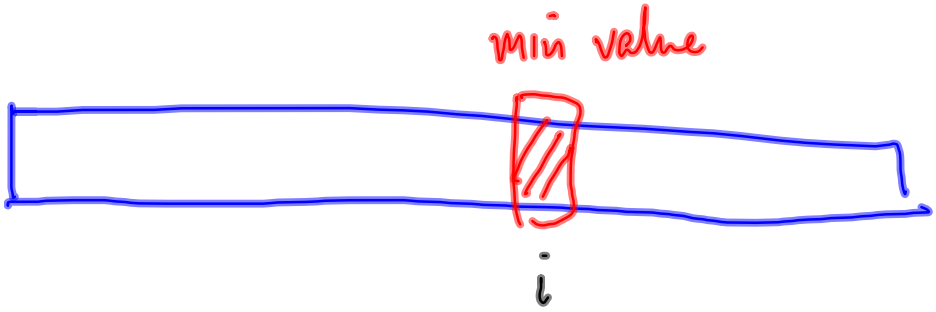
⇓

"extract minimum"

```
def selection_sort(l):
    if l == []:
        return l
    else:
        x = extract minimum(l)
        y = selection_sort(rest of l)
        return [x] + y
```


l [ ]
[x0] [ ]
+ [x1] [ ]
+ . . .
[xn-1]
+ [xn-1] + [ ]

What should extract_minimum do?



1. del(a[i]) → rest of l is l itself
2. Swap a[i] & a[0] → rest of l is l[1:]

```
def extract_minimum(l):
    minpos = 0
    for i in range(1, len(l)):
        if l[i] < l[minpos]:
            minpos = i
    minval = l[minpos]
    del(l[minpos])
    return(minval)
```

```
def selection_sort(l):
    if l == []:
        return(l)
    else:
        m = extract_minimum(l)
        return([m] + selection_sort(l))
```

l is updated in place, loses min value

i.e. rest of l

```
def extract_minimum(l):
    minpos = 0
    for i in range(1, len(l)):
        if l[i] < l[minpos]:
            minpos = i
    (l[0], l[minpos]) = (l[minpos], l[0])
    return() # optional, function does not need to return a value
```

Also Update l in place

```
def selection_sort(l):
    if l==[]:
        return(l)
    else
        extract_minimum(l)
        return ( l[:1] + selection_sort(l[1:]))
                 [l[0]]
```

l = selection_sort(l)

even if l is modified internally by extract_minimum, we finally reassign l to the new list (sorted)

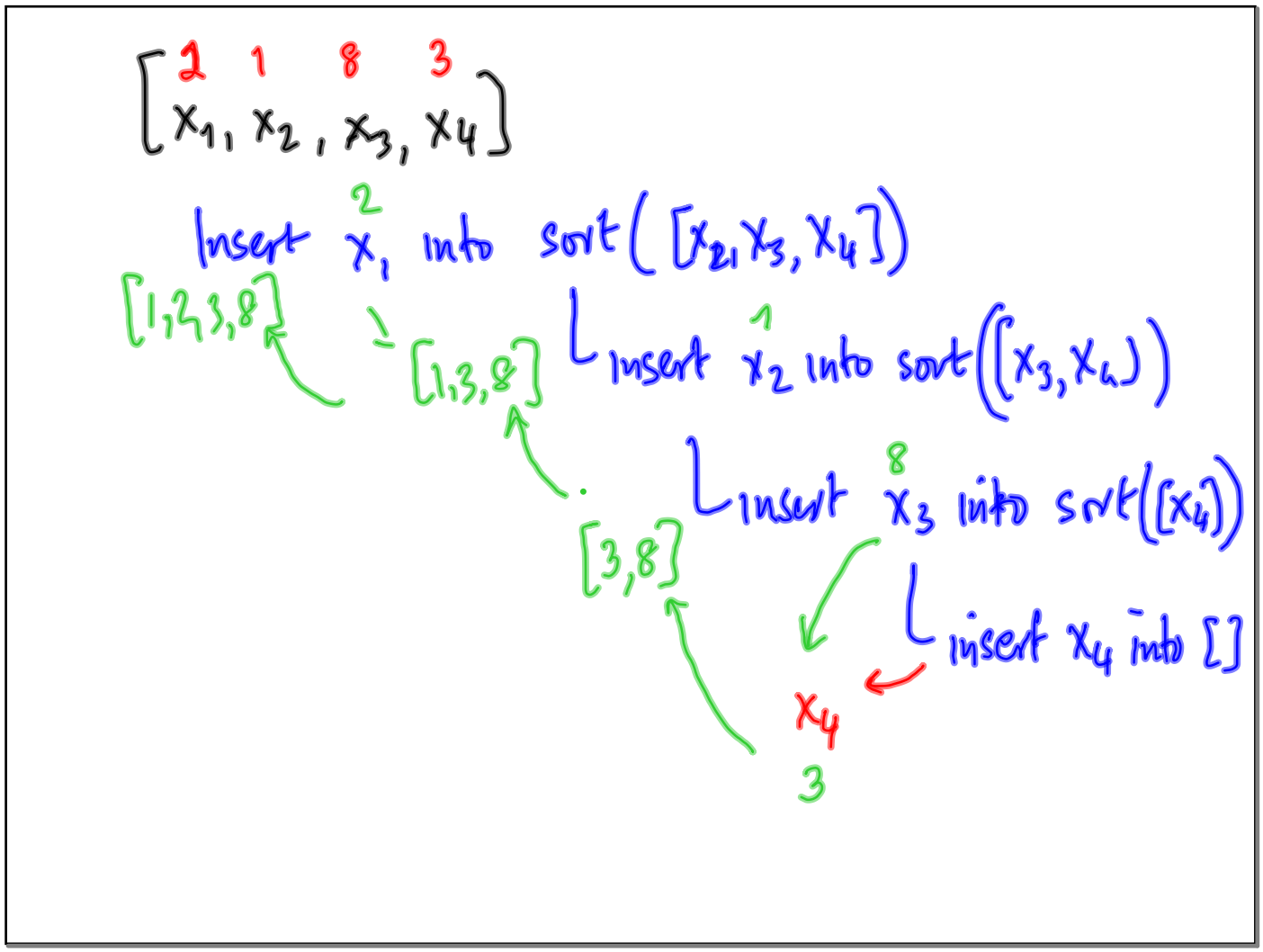
Another way to sort?

Build a sorted list

Insert next item in place

Insertion Sort

```
def insertion_sort(l):
    if l == []:
        return(l)
    else
        Inductively sort l[1:]
        Insert l[0] into this sorted list
```


[2 1 8 3]
x1, x2, x3, x4
Insert x1 into sort([x2, x3, x4])
[1,2,3,8]
[1,3,8]
Insert x2 into sort([x3, x4])
[3,8]
Insert x3 into sort([x4])
Insert x4 into []
x4
3

```
def insert(x, l):          # l is sorted, ascending
    # Naive soln: find correct position to insert x
    if l == []:
        return([x])
    elif x < l[0]:
        return([x] + l)
    else:                  # [l[0]]
        return(l[:1] + insert(x, l[1:]))
```

```
def insertion_sort(l):
    if l == []:
        return(l)
    else
        return(insert(l[0], insertion_sort(l[1:])))
```

Efficiency?

Count the no. of steps of f(n) as a function of size(n).

Sorting

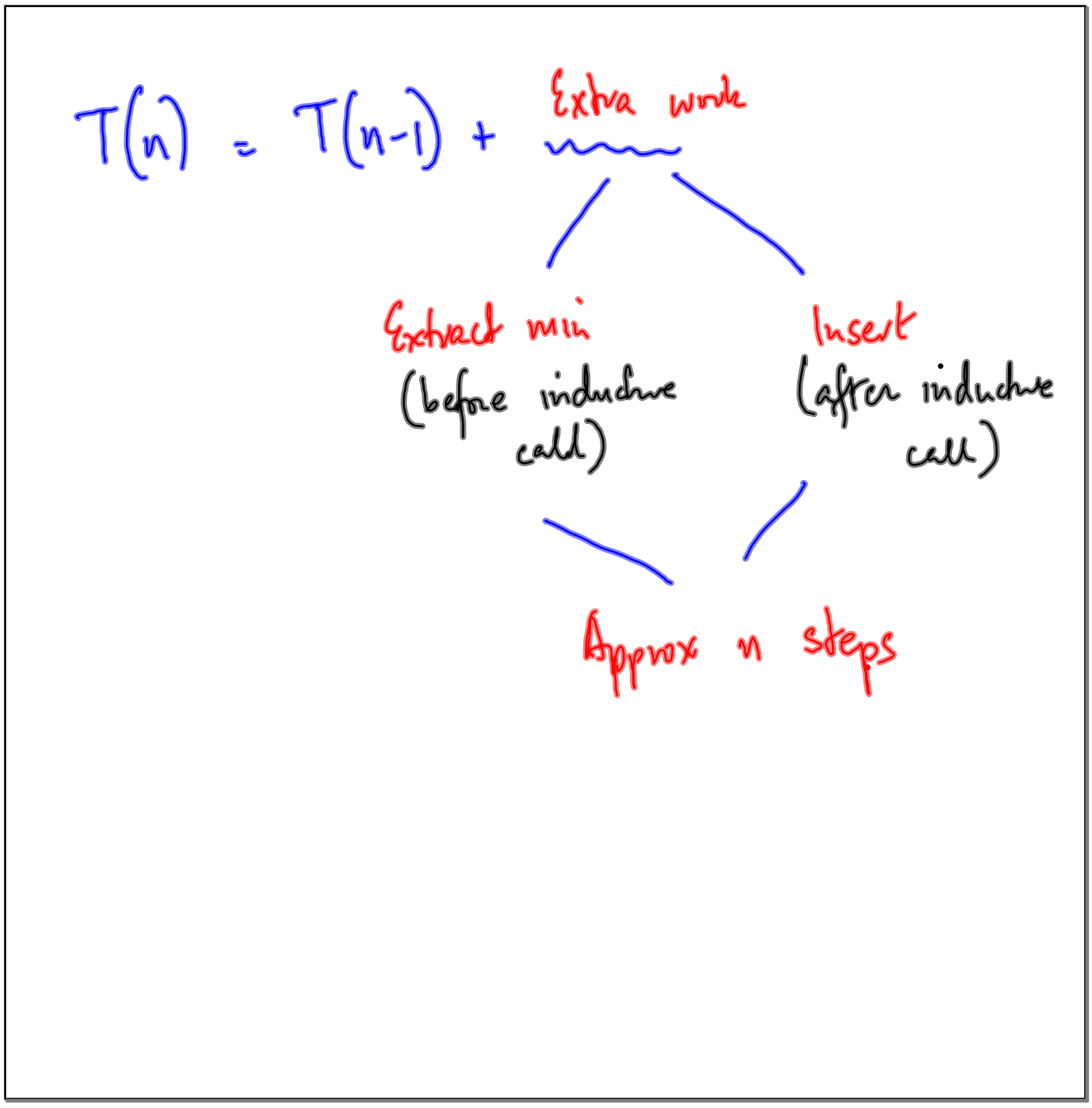
Count comparisons ($x[i] \overset{?}{<} x[j]$)

and movements ($(x,y) = (y,x)$)

Let $T(n)$ denote time taken on an input of size n

Insertion or selection sort

Input of size n → Solve for size $n-1$
Do some work to set up problem of size $n-1$, combine answer


T(n) = T(n-1) + Extra work
Extract min
(before inductive call)
Insert
(after inductive call)
Approx n steps