

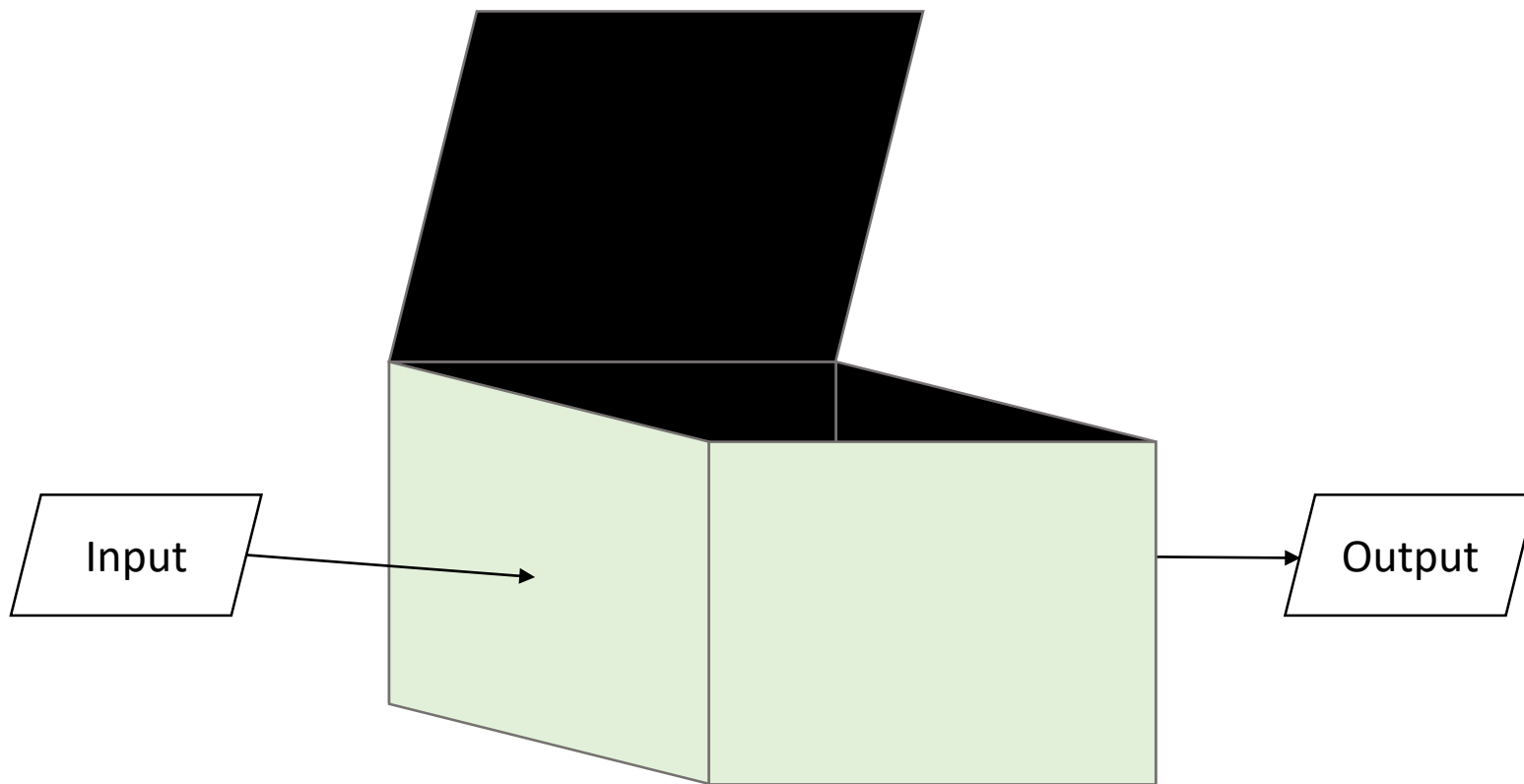
Lecture 3

Decrease and Conquer algorithms

Text sections 4.1, 4.3, 4.4

Aside:

Programs/algorithms
as “black boxes”



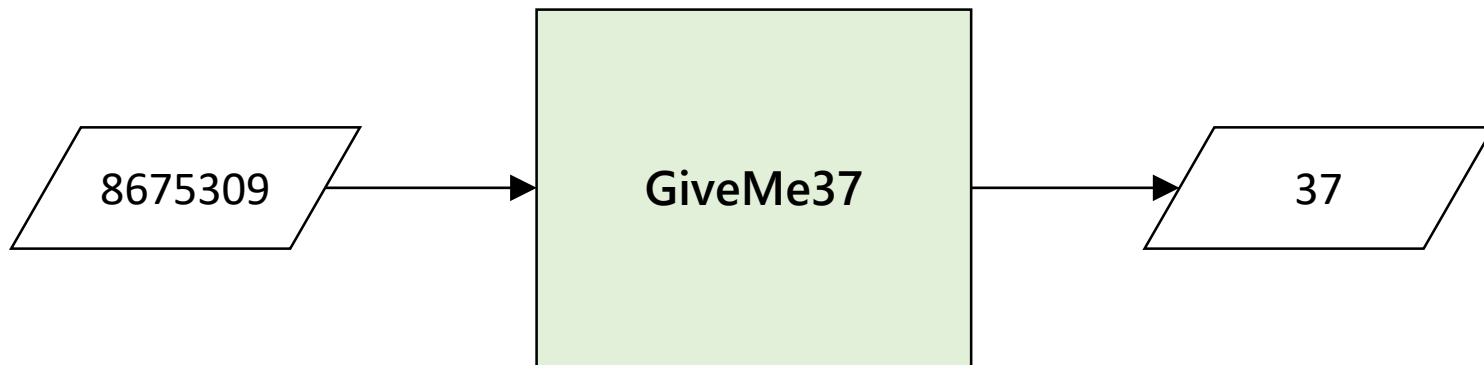
Example: program to output “37”

- Input: anything (or nothing!)
- Output: 37

```
Algorithm GiveMe37()  
    return 37  
END
```

```
public class GiveMe37 {  
    public static void main(String[] args) {  
        System.out.println(37);  
    }  
}
```

Previous program as a black box*

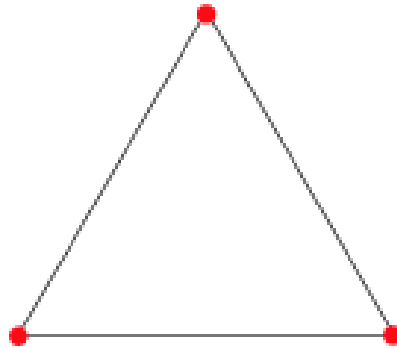


* Well, a green box in this case.

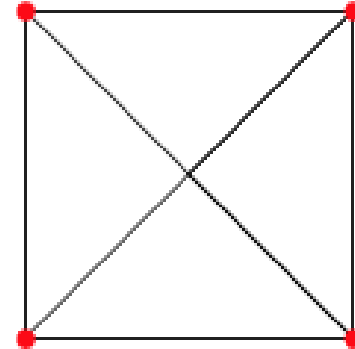
Q: How many edges in a *complete graph*?



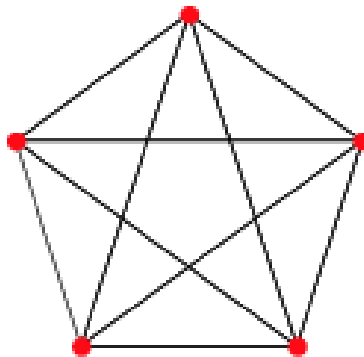
K_2



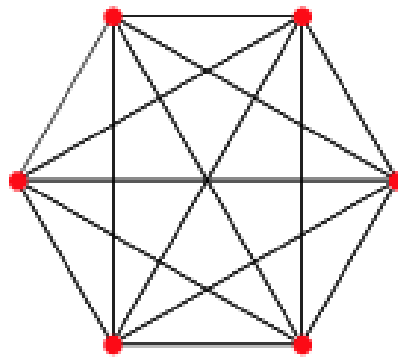
K_3



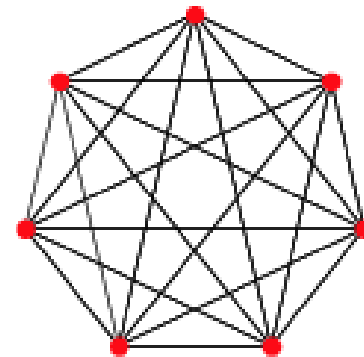
K_4



K_5



K_6



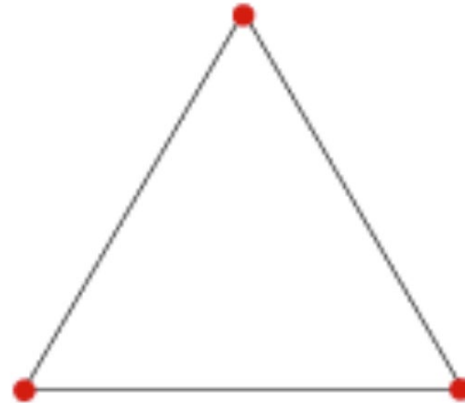
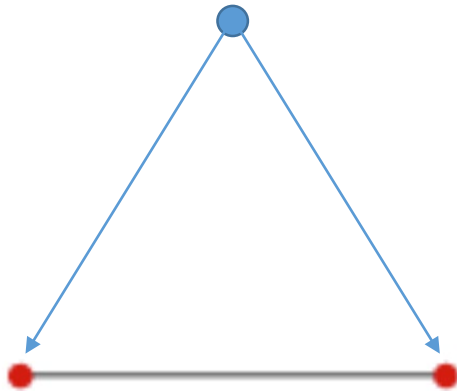
K_7

Relationship between K_n and K_{n+1}

- Add one vertex
- Connect it to (all) n other vertices (add n edges)

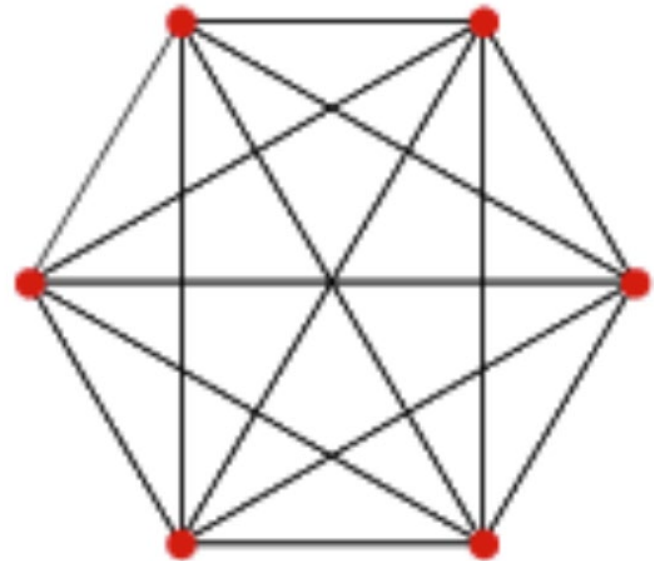
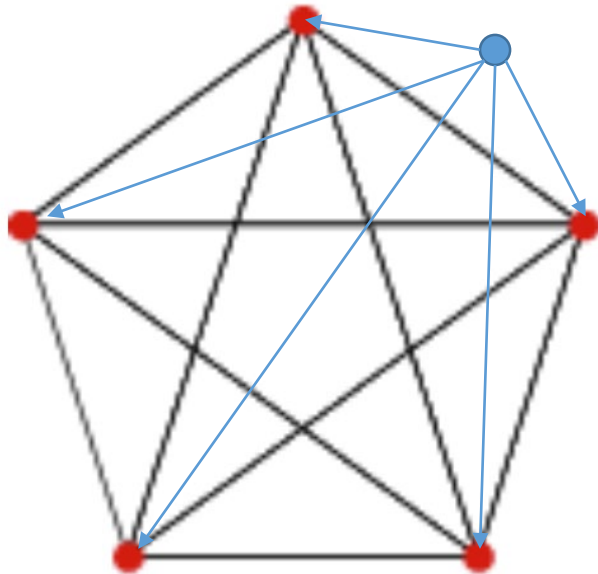
Constructing K_3 from K_2

1. Add one vertex
2. Connect it to (all)(2) other vertices



Constructing K_6 from K_5

- Add one vertex
- Connect it to (all)(5) other vertices



Q: How many edges in a complete graph K_n ?

- A: *If only we knew* the answer for K_{n-1} ...
- ... then we could get the answer for K_n

How many edges in K_n ?

- Recursive definition (algorithm):

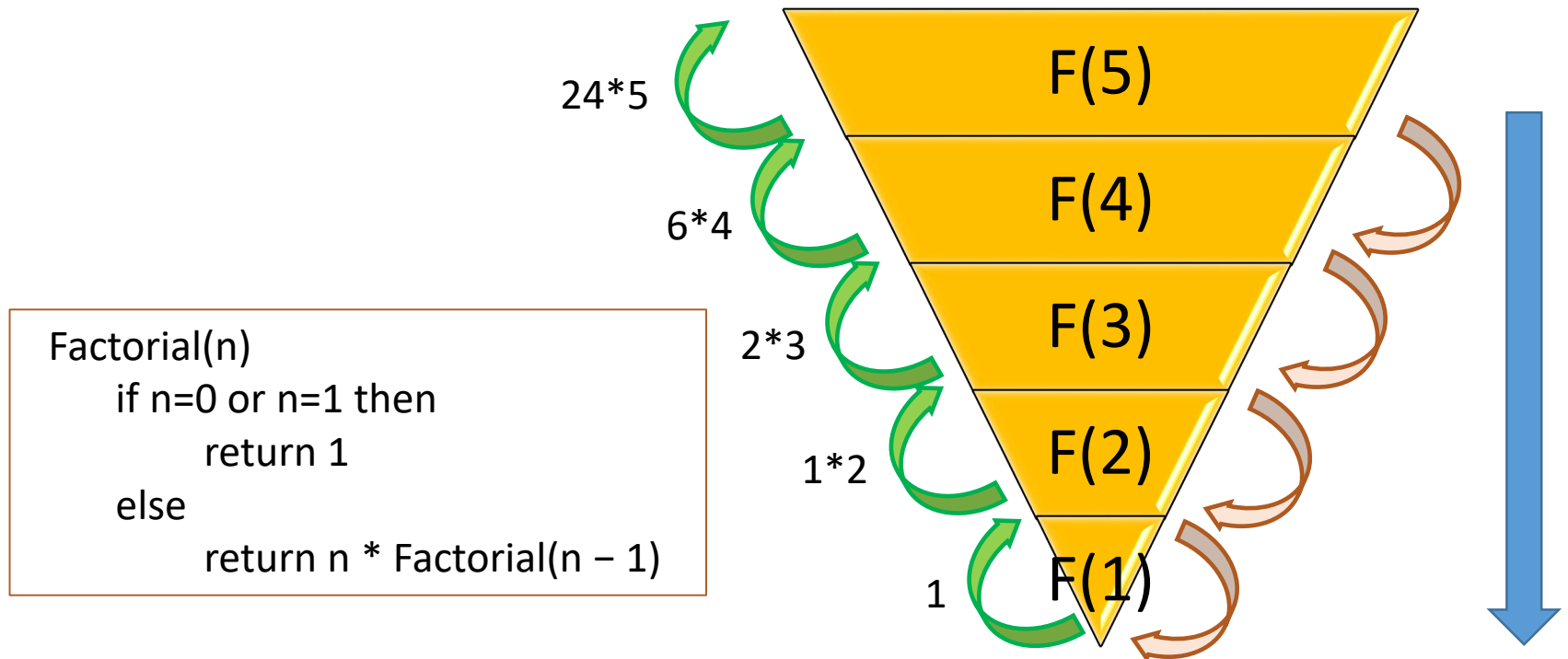
```
ALGORITHM num_edges(int n)
// n is the number of vertices in a complete graph
// Return the number of edges in the graph
    if n = 1
        return 0
    else
        return (n-1) + num_edges(n-1)
    endif
END
```

- $\text{num_edges}(K_{37}) = 36 + \text{num_edges}(K_{36})$

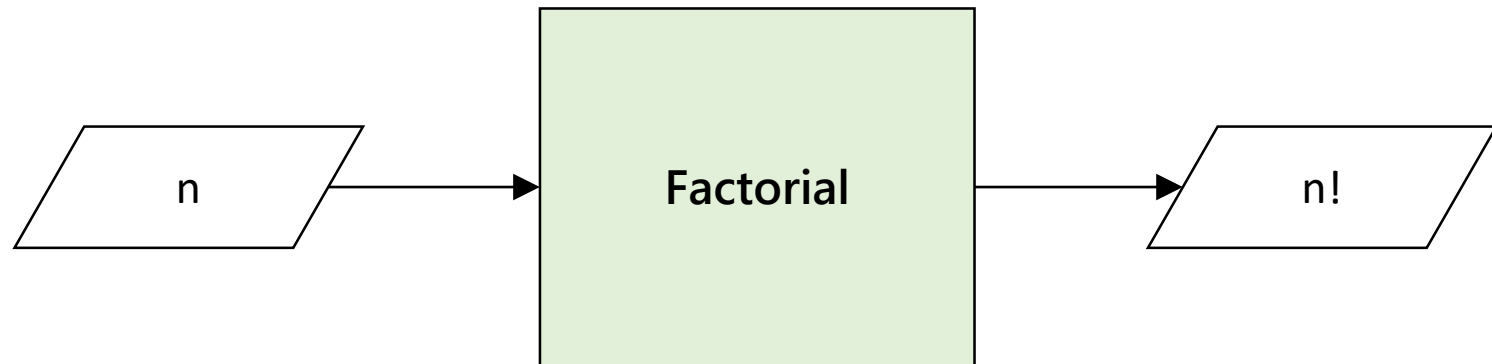
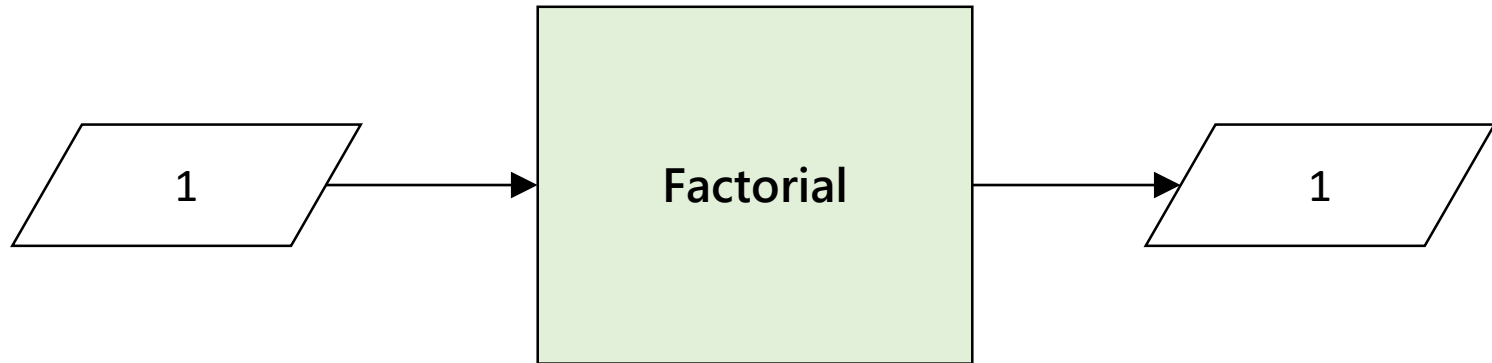
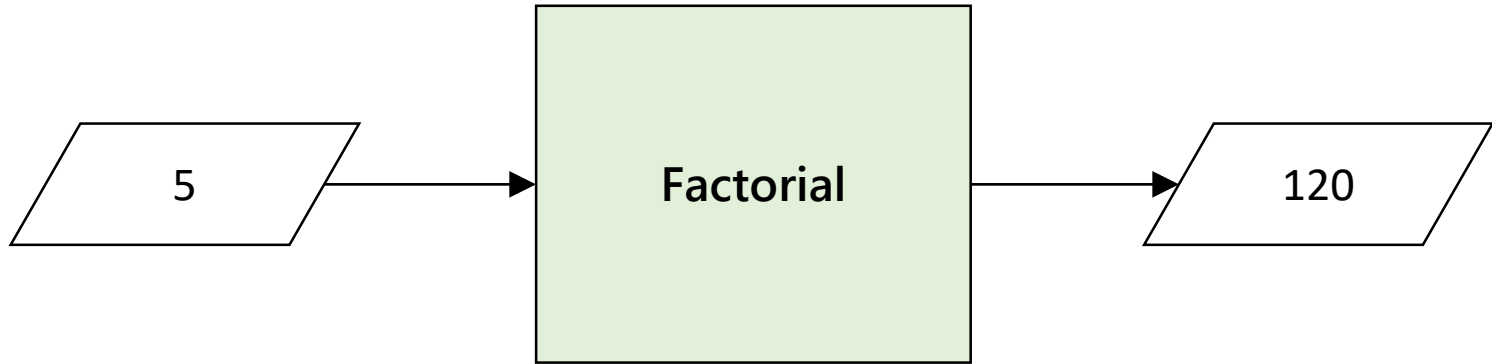
Decrease and conquer

- Reduce problem instance to smaller instance of the same problem and solve smaller instance
 - I.e. ***Solve a smaller problem***
- Extend solution of smaller instance to obtain solution to original instance
 - Extend, augment, enhance, adapt, adjust, ...
 - Sometimes this part is trivial
- Can be implemented:
 - Top-down (recursive)
 - Bottom-up (iterative)

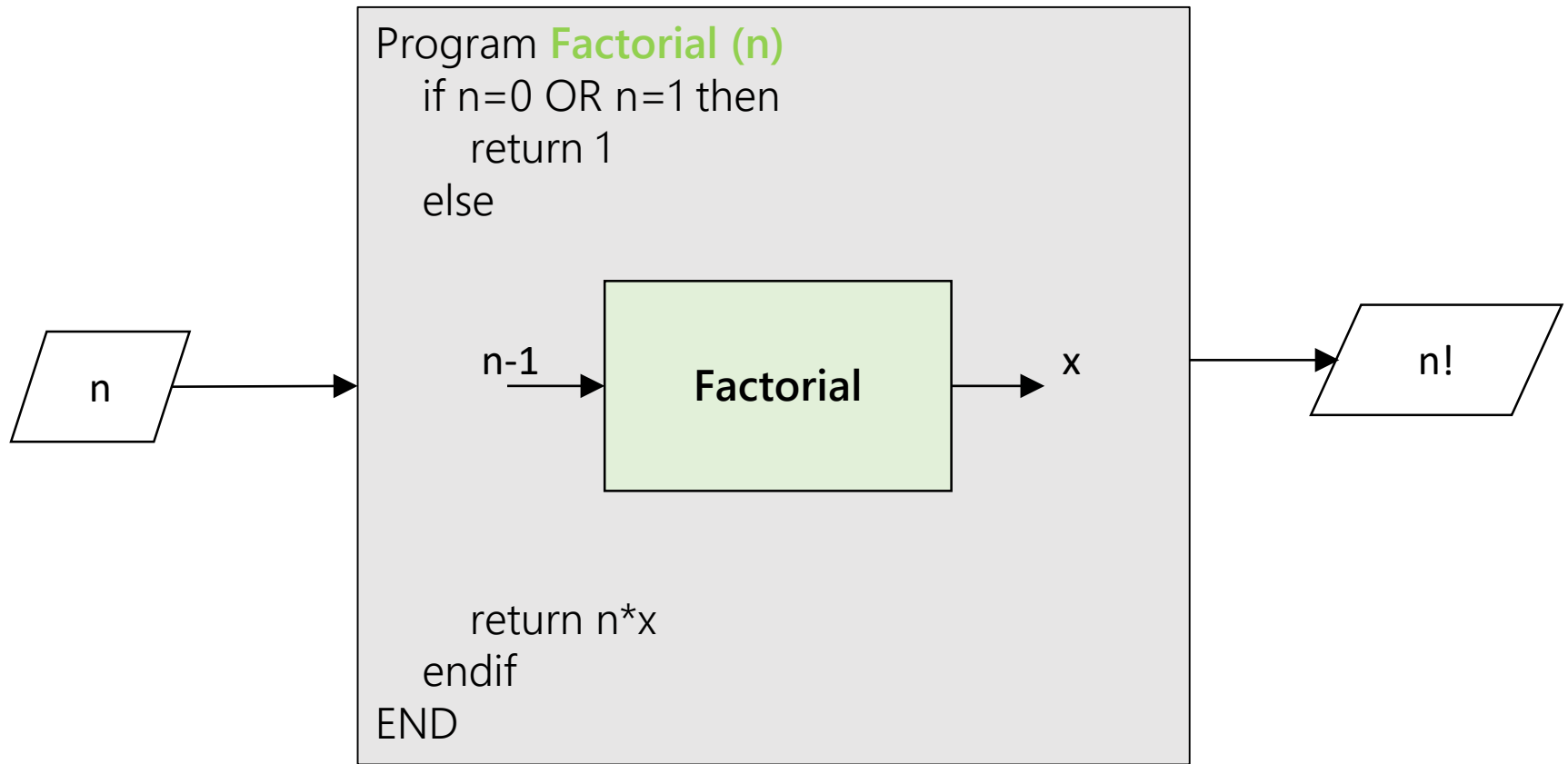
Example: top-down (recursive)



Factorial (5)= ?



Inside the box



Example: bottom-up (iterative)

Factorial (n)

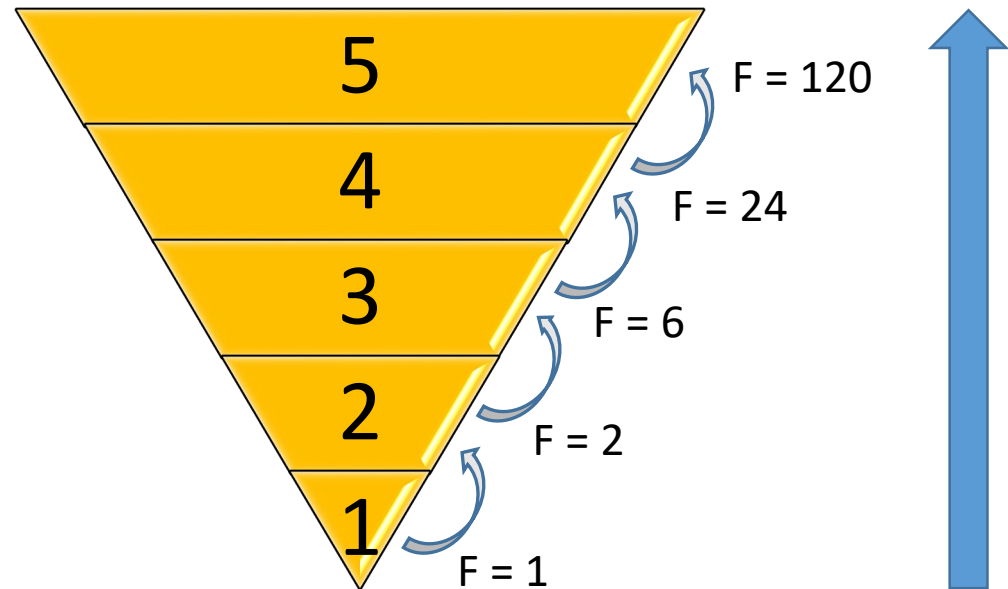
$F \leftarrow 1$

for $i \leftarrow 1$ to n

$F \leftarrow F * i$

return F

Factorial (5) = ?

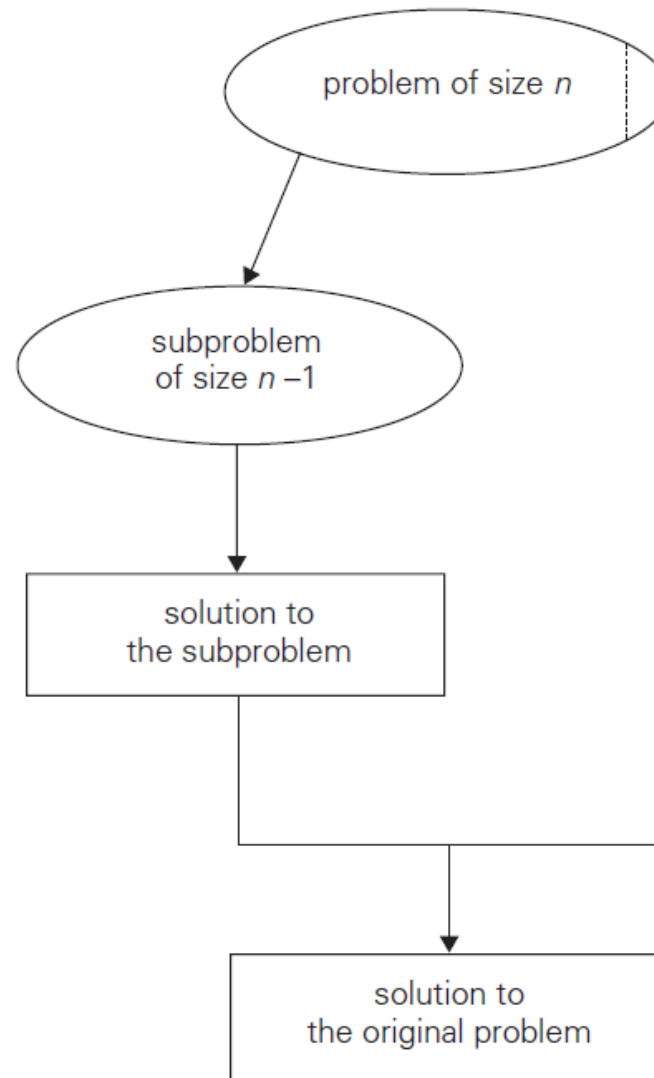


Three types of Decrease and Conquer

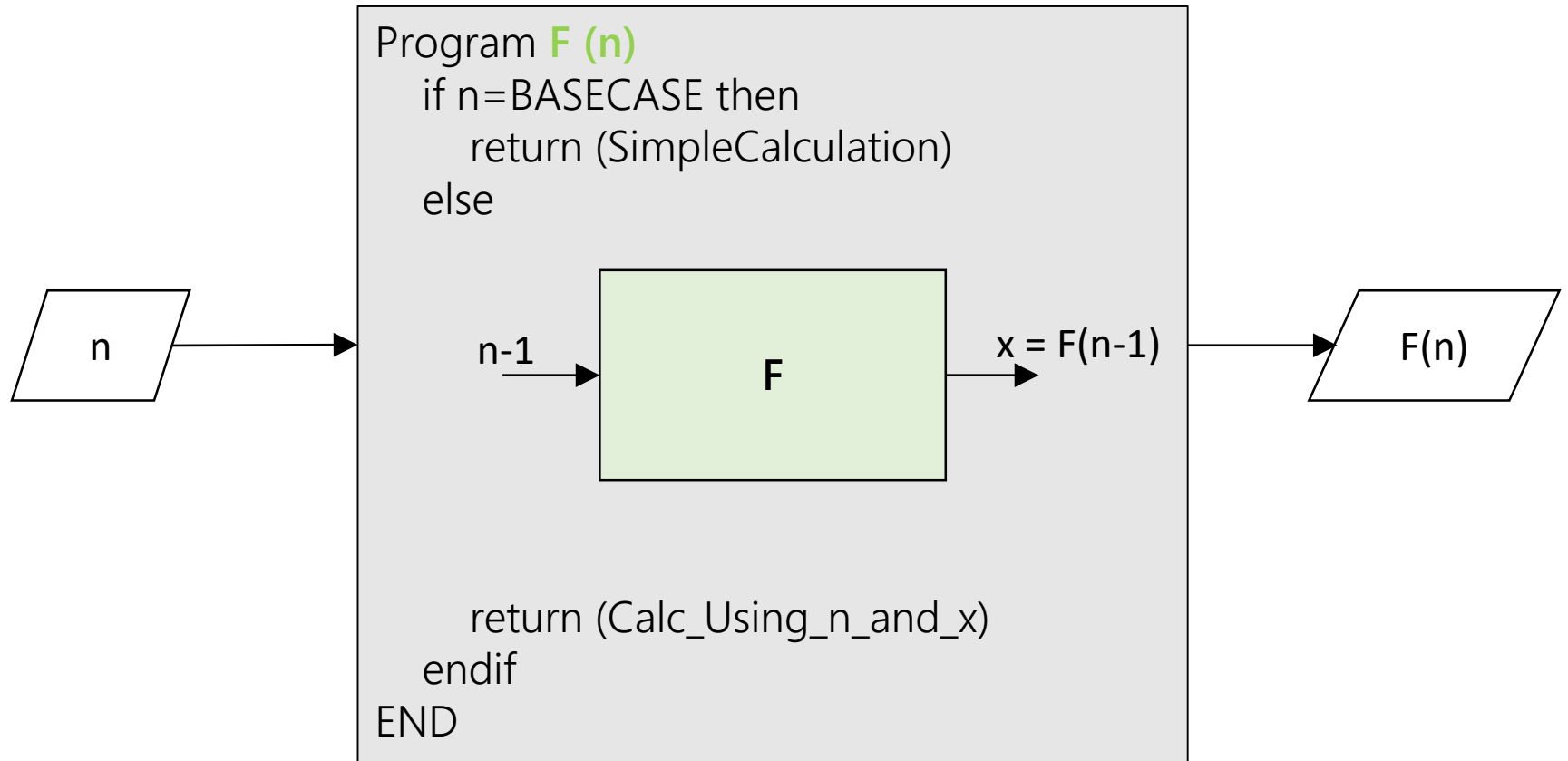
- Decrease by a constant (usually by 1)
 - Insertion sort
 - Generating permutations
 - Generating subsets
- Decrease by a constant factor (usually by half)
 - Binary search
 - Exponentiation by squaring
 - Fake coin problem
- Variable-size decrease
 - Euclid's algorithm

Decrease by a
constant amount

Decrease by a constant (often 1)



Inside the box of “decrease by constant amount”



Generating permutations

Example of “decrease by 1”

Permutations of N objects

N=1

A

N=2

AB

BA

N=3

ABC

ACB

BAC

BCA

CAB

CBA

N=4

ABCD BACD CABD DABC

ABDC BADC CADB DACB

ACBD BCAD CBAD DBAC

ACDB BCDA CBDA DBCA

ADBC BDAC CDAB DCAB

ADCB BDCA CDBA DCBA

N=4, hide the Ds

ABC. BAC. CAB. .ABC

AB.C BA.C CA.B .ACB

ACB. BCA. CBA. .BAC

AC.B BC.A CB.A .BCA

A.BC B.AC C.AB .CAB

A.CB B.CA C.BA .CBA

Generating permutations

- Example: To find all permutations of 3 objects A, B, C
 - First find all permutations of 2 objects, say B and C:

B C and C B

- Then insert the remaining object, A, into *all possible positions* in each of the permutations of B and C:

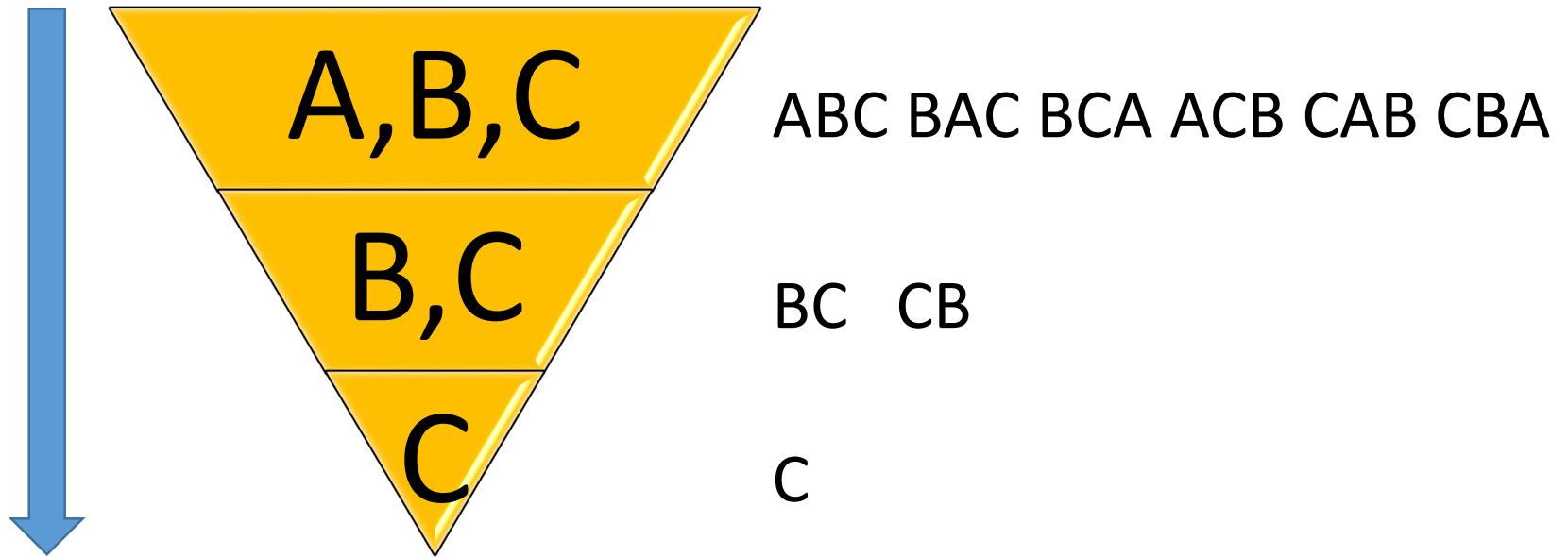
A B C B A C B C A and A C B C A B C B A

Generating permutations

- To find all permutations of n objects:
 1. Find all permutations of $n-1$ of those objects
 2. Insert the remaining object into all possible positions of each permutation of $n-1$ objects

Generating permutations

- Example: find all permutations of A, B, C



Generating permutations

```
generatePerms ( $a_1, a_2, \dots, a_n$ )  
  if  $n > 1$   
    smallerPerms = generatePerms ( $a_1, a_2, \dots, a_{n-1}$ )  
    initialize allPerms to {}  
    for each p in smallerPerms  
      insert  $a_n$  before  $a_1$  and add to allPerms  
      for  $i \leftarrow 1$  to  $n-1$   
        insert  $a_n$  after  $a_i$  and add to allPerms  
  return allPerms
```

Generating subsets

Example of “decrease by 1”

Subsets of $\{a,b,c,d\}$

In “lexicographic” order:

$\{\}$,
 $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$,
 $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{b,c\}$, $\{b,d\}$, $\{c,d\}$,
 $\{a,b,c\}$, $\{a,b,d\}$, $\{a,c,d\}$, $\{b,c,d\}$,
 $\{a,b,c,d\}$

Let's rearrange them a little:

$\{\}$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
↕	↕	↕	↕	↕	↕	↕	↕
$\{d\}$	$\{a,d\}$	$\{b,d\}$	$\{c,d\}$	$\{a,b,d\}$	$\{a,c,d\}$	$\{b,c,d\}$	$\{a,b,c,d\}$

All the
sets
without d

All the
sets
with d

Generating subsets: IDEA



To find all subsets of a set with N items:

1. Find all subsets of a set with $N-1$ of the items
2. Copy/clone the subsets
3. Insert the last item into all the copies

Subsets of $A = \{a, b, c, d, e, \dots, z\}$

1

Find subsets
of $A - \{z\}$
(smaller problem!)

$\{\}$ $\{a\}$
 $\{b\}$ $\{c\}$
 $\{a, b\}$ $\{a, c\}$
 $\{a, d\}$ $\{a, e\}$
... $\{a, b, c\}$
 $\{a, b, d\}$...
 $\{c, d, e\}$...
 $\{a, b, c, d\}$
... $\{\dots\}$...
 $\{\dots\}$

2

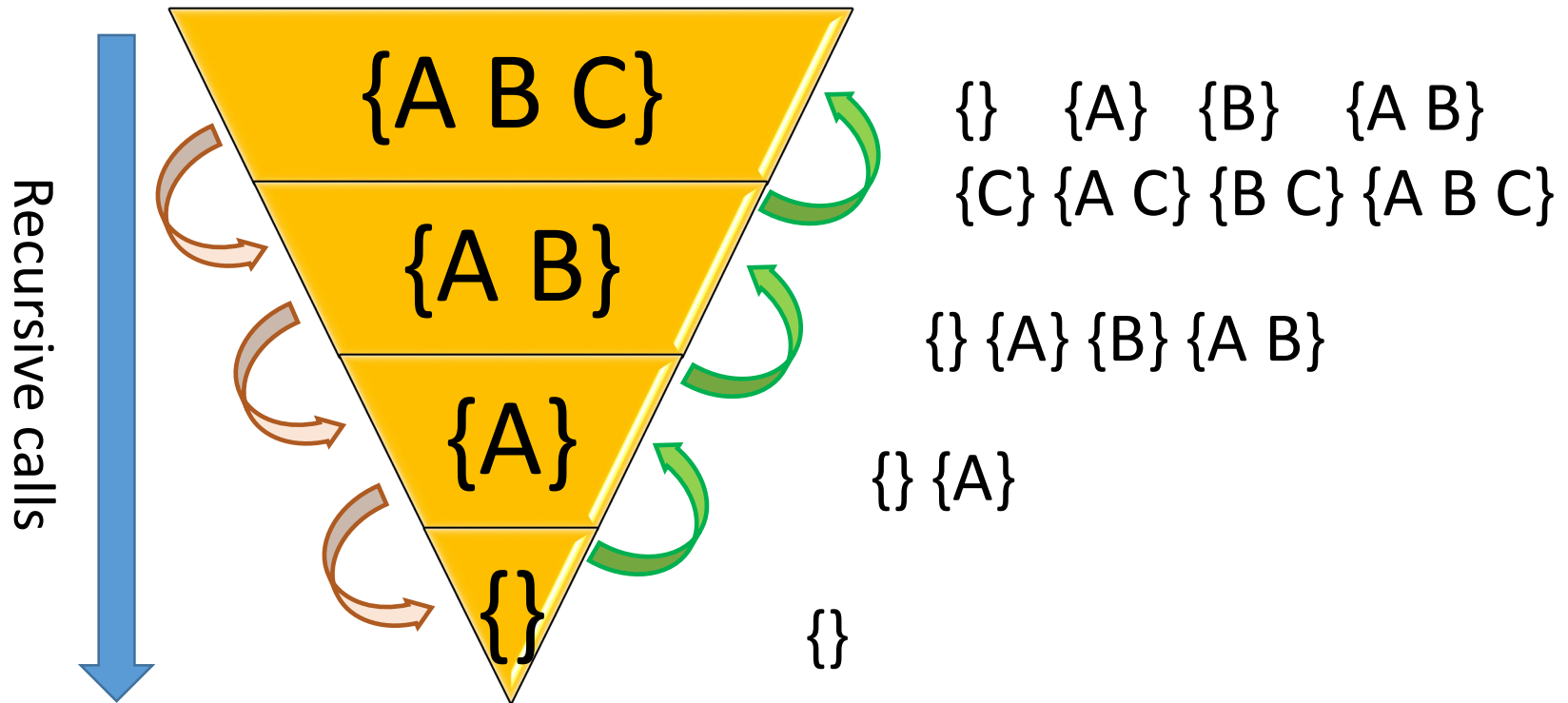
Duplicate the result, and add
 z to every set in the copy

$\{\}$ $\{a\}$
 $\{b\}$ $\{c\}$
 $\{a, b\}$ $\{a, c\}$
 $\{a, d\}$ $\{a, e\}$
... $\{a, b, c\}$
 $\{a, b, d\}$...
 $\{c, d, e\}$...
 $\{a, b, c, d\}$
... $\{\dots\}$...
 $\{\dots\}$

$\{z\}$ $\{a, z\}$
 $\{b, z\}$ $\{c, z\}$
 $\{a, b, z\}$ $\{a, c, z\}$
 $\{a, d, z\}$ $\{a, e, z\}$
... $\{a, b, c, z\}$
 $\{a, b, d, z\}$...
 $\{c, d, e, z\}$...
 $\{a, b, c, d, z\}$
... $\{\dots, z\}$...
 $\{\dots, z\}$

Generating subsets

- Example: find all subsets of $\{A, B, C\}$



Generating subsets

```
generateSubsets ( $a_1, a_2, \dots, a_n$ )  
  if  $n > 0$   
    subsets = generateSubsets ( $a_1, a_2, \dots, a_{n-1}$ )  
    for each subset  $s$  in subsets  
      clone  $s$  to  $s'$   
      insert  $a_n$  to  $s'$   
      add  $s'$  to subsets  
  return subsets
```


Insertion sort

Example of “decrease by 1”

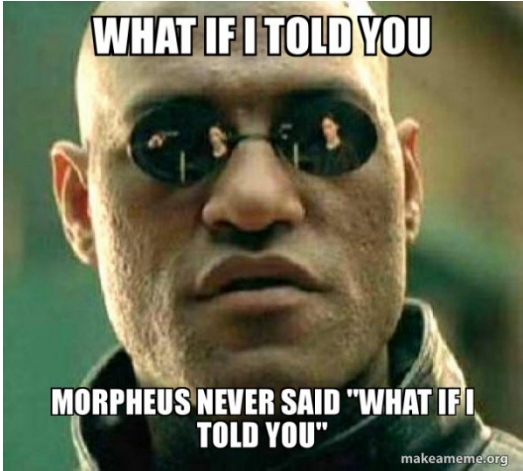
A close-up of Morpheus from The Matrix, wearing his signature black sunglasses. The reflection in the sunglasses shows a scene from the movie. The text "WHAT IF I TOLD YOU" is overlaid at the top in a bold, white, sans-serif font with a black outline.

WHAT IF I TOLD YOU

**ALL EXCEPT THE LAST ITEM WAS
SORTED ALREADY?**

makeameme.org

Then “sort”
would just
be “shift
over some
items and
drop $A[n-1]$
into place”.

A smaller version of the Morpheus meme image. The text "WHAT IF I TOLD YOU" is overlaid at the top in a bold, white, sans-serif font with a black outline.

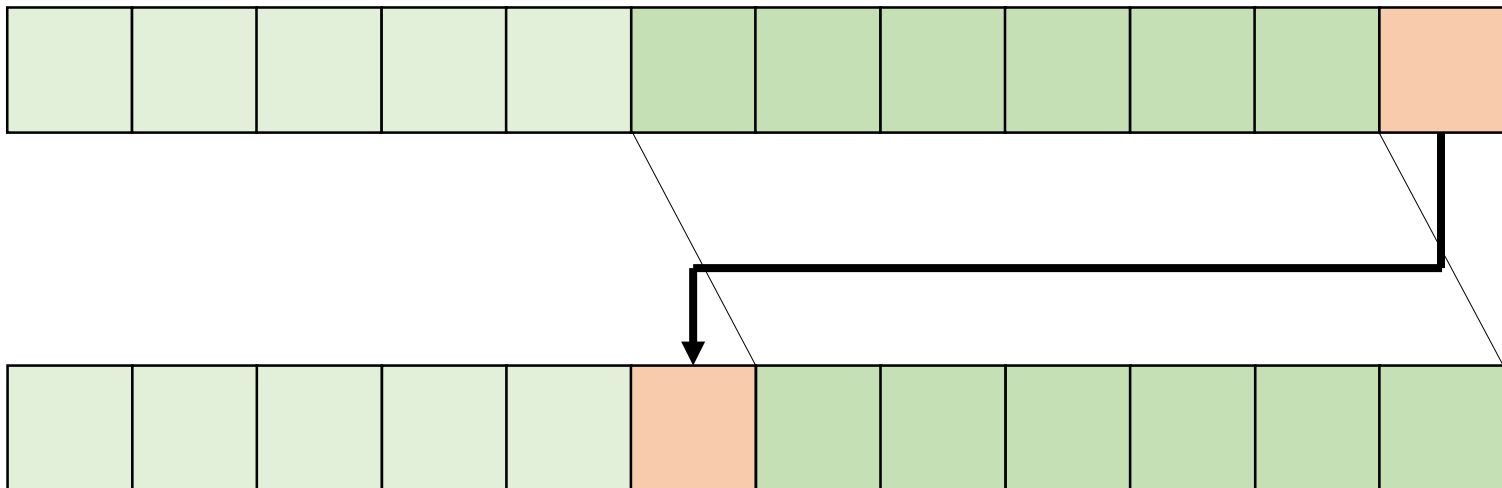
WHAT IF I TOLD YOU

**MORPHEUS NEVER SAID "WHAT IF I
TOLD YOU"**

makeameme.org

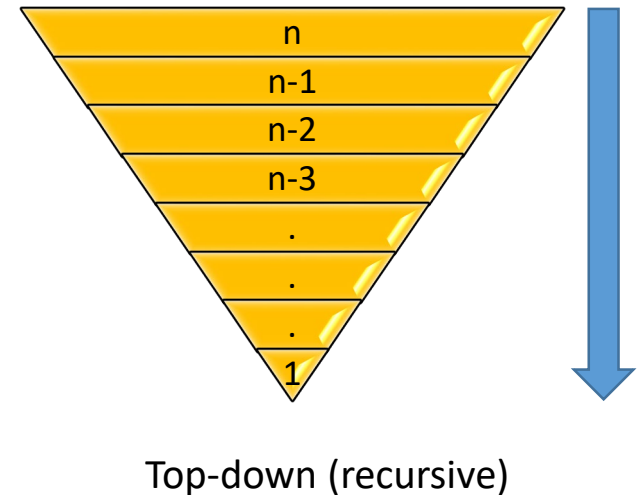
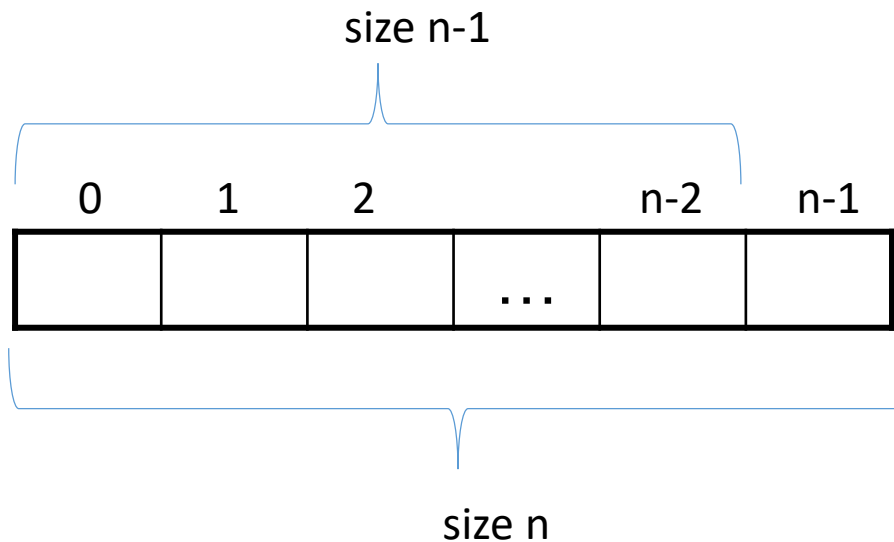
Sort algorithm idea:

1. Sort items $A[0]$ through $A[n-2]$
 - This is a *big* step ... think of it as a subroutine
2. Find the spot where last item $A[n-1]$ goes
3. Shift items over and drop in $A[n-1]$

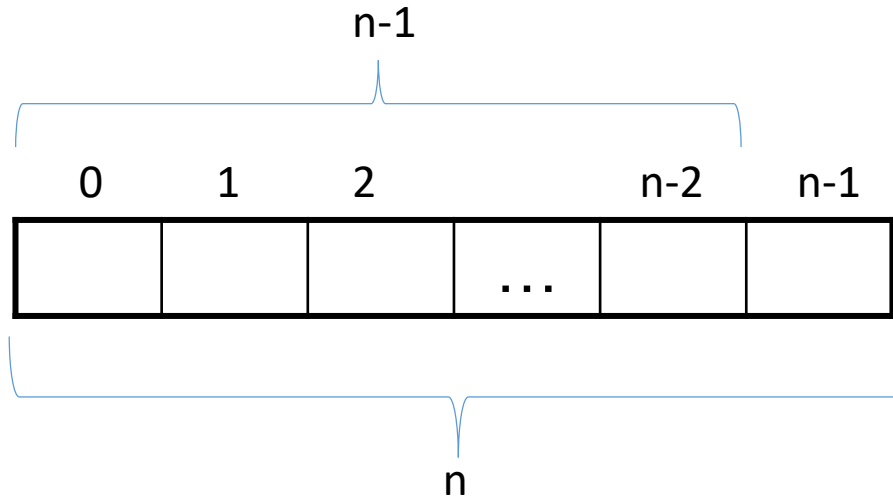


Insertion sort

- Insertion sort ($A[0..n-1]$)
 1. Sort $A[0..n-2]$
 2. Insert $A[n-1]$ in its proper place among the sorted $A[0..n-2]$



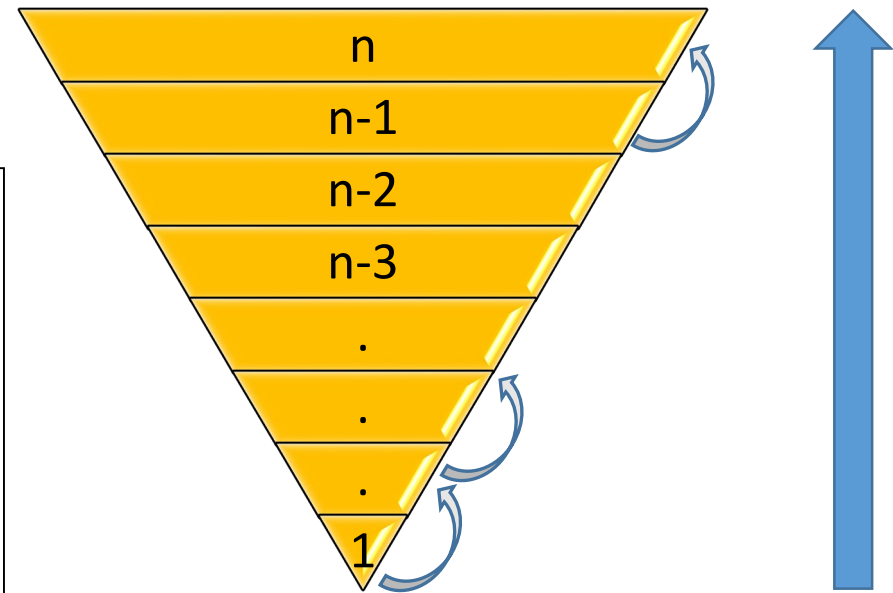
Insertion sort (recursive)



```
InsertionSort(A,n)
1  if n > 1
2      InsertionSort(A,n-1)
3      key  $\leftarrow$  A[n-1]
4      i = n-2
5      while i  $\geq$  0 and A[i] > key
6          A[i+1]  $\leftarrow$  A[i]
7          i  $\leftarrow$  i - 1
8      A[i + 1]  $\leftarrow$  key
```

Insertion sort (iterative)

```
1. InsertionSort(A[0..n-1])
2.   for i ← 1 to n-1 do
3.     v ← A[i]
4.     j ← i-1
5.     while j ≥ 0 and A[j] > v do
6.       A[j+1] ← A[j]
7.       j ← j-1
8.     A[j+1] ← v
```



bottom-up

Insertion sort and Selection sort: Similarities

- "Sorted" and "unsorted" piles
- Each main iteration does two things:
 - Choose item from "unsorted"
 - Place item in "sorted"
- Number of main iterations is $O(n)$
- $O(n^2)$ overall (worst case)

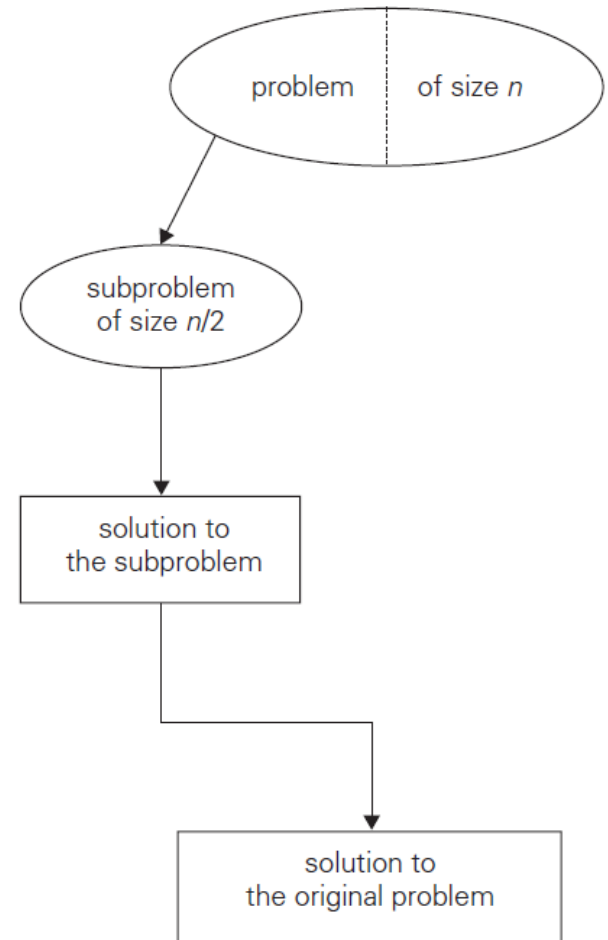
Insertion sort and Selection sort: Differences

- Selection sort: each main iteration
 - "Choose from unsorted part" is $O(n)$ (linear search)
 - "Place into sorted part" is $O(1)$ (it goes at the end)
- Insertion sort: each main iteration
 - "Choose from unsorted part" is $O(1)$ (choose first item)
 - "Place into sorted part" is $O(n)$ (shift the other items)

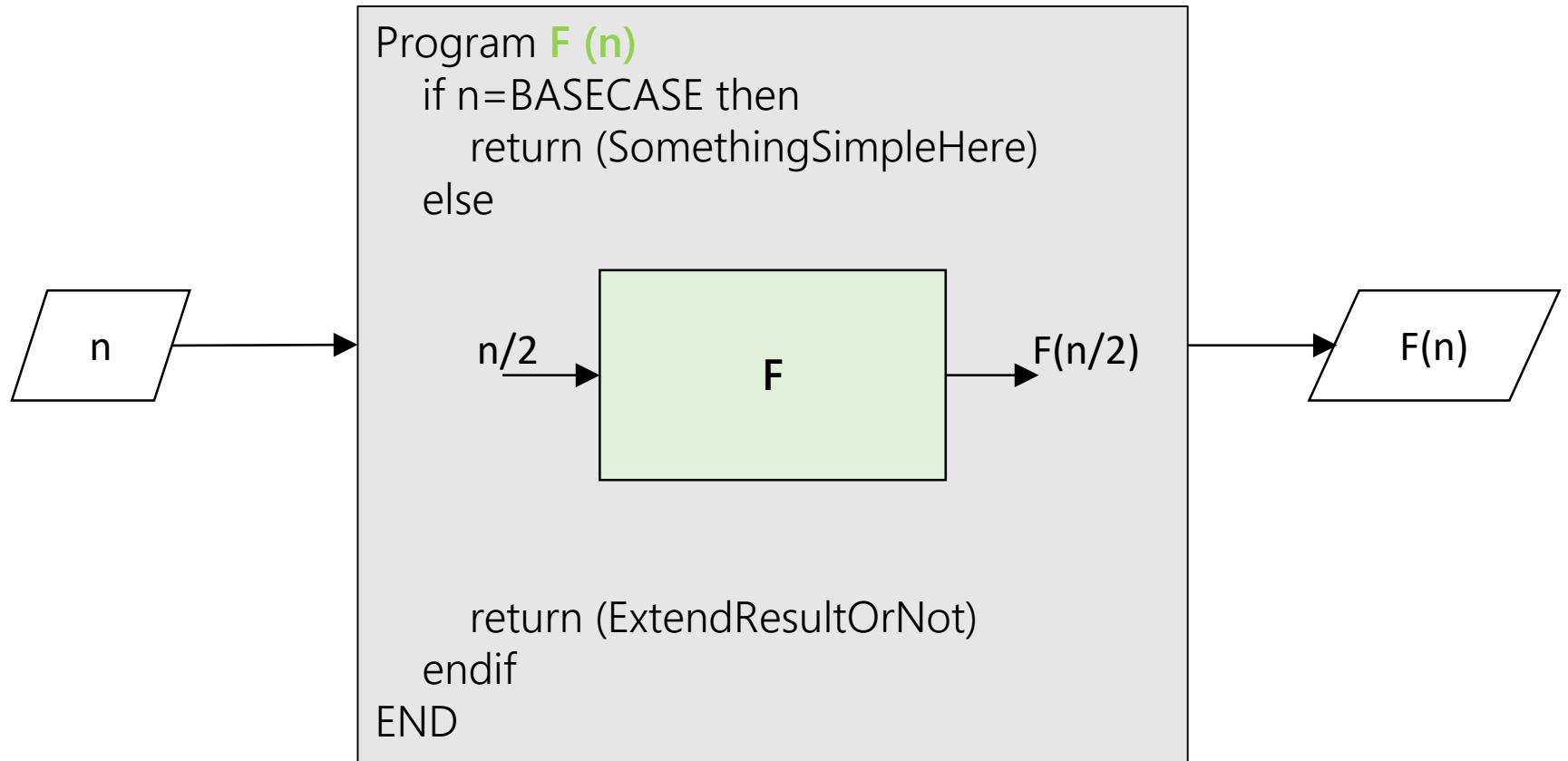
Decrease by a
constant factor

Decrease by a constant factor

- Make the problem smaller by some constant factor
- Often the constant factor is *two*, i.e, we divide the problem in half
- Discard one or more of the parts



Inside the box of “decrease by constant factor”



Binary search

Example of “decrease by factor of 2”

i.e. solve a problem of size $n/2$

Binary search

- Example: binary search, key = 7

Sorted
Array

3	6	7	11	32	33	53
---	---	---	----	----	----	----

3	6	7	11	32	33	53
---	---	---	----	----	----	----

3	6	7
---	---	---

7

Binary search

- Binary Search works by dividing the sorted array (i.e. the *solution space*) in half each time, and searching in the half where the target should exist
- In other words, we eliminate half the input on each iteration!
- It makes efficiency gains by ignoring the part of the solution space that we know cannot contain a feasible solution

Binary search

```
binarySearch(a[], k, s, e)
if e < s
    return not found
m ← floor((s+e)/2)
if k > a[m]
    return binarySearch(a[], k, m+1, e)
else if k < a[m]
    return binarySearch(a[], k, s, m-1)
else
    return m
```

Binary search

binarySearch(a[], k, s, e)

- Example: Binary search, k=90

<i>index</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	6	7	9	9	13	17	22	25	41	43	47	61	62	64	78	81	88	90	91	92	93

Binary search

binarySearch(a[], k, s, e)

- Example: Binary search, k=90

<i>index</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	6	7	9	9	13	17	22	25	41	43	47	61	62	64	78	81	88	90	91	92	93

Call trace:

1. *binarySearch(a, 90, 0, 20)*

1.1 *binarySearch(a, 90, 11, 20)*

1.1.1 *binarySearch(a, 90, 16, 20)*

1.1.1.1 *binarySearch(a, 90, 16, 17)*

1.1.1.1.1 *binarySearch(a, 90, 17, 17)*

***target found, returns*

Binary search efficiency

- Time efficiency
 - Worst-case efficiency...
 - $C(n) = \log_2(n) + 1$
 - So binary search is $O(\log n)$
 - This is VERY fast: e.g., $C(1000000) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array

Binary search (recursive)

Example: Trace the values of s,e,m as the algorithm runs with different keys (k)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	6	7	9	9	13	17	22	25	41	43	47	61	62	64	78	81	88	90	91	92	93

- Trace for k=81 (s=0, e=20 initially)
 - iteration 1: s,e,m = 11,20,10
 - iteration 2: s,e,m = -, -,15 ** target found
- Trace for k=22
 - iteration 1: s,e,m = 0,9,10
 - iteration 2: s,e,m = 5,9,4
 - iteration 3: s,e,m = 5,6,7
 - iteration 4: s,e,m = 6,6,5
 - iteration 5: s,e,m = -, -,6 ** target found
- Note: largest number of iterations is 6, when the target is not found in the array being searched (generally it will be $\lceil \log_2 n \rceil + 1$)

Binary search (iterative)

```
binarySearch(a[], s, e, k)
```

```
while  $s \leq e$ 
```

```
     $m \leftarrow \text{floor}((s+e)/2)$ 
```

```
    if  $k > a[m]$ 
```

```
         $s \leftarrow m+1$ 
```

```
    else if  $k < a[m]$ 
```

```
         $e \leftarrow m-1$ 
```

```
    else
```

```
        return  $m$ 
```

```
return not found
```

Exponentiation by squaring

Example of “decrease by factor of 2”

i.e. solve problem of size $n/2$

Exponentiation by squaring

- Compute a^n where n is a nonnegative integer
- Brute-force algorithm requires $n-1$ multiplications
- We can do much better!

Example: calculating a^{38}

$$a^{38} \rightarrow a^{19} * a^{19}$$

$$a^{19} \rightarrow a * a^9 * a^9$$

$$a^9 \rightarrow a * a^4 * a^4$$

$$a^4 \rightarrow a^2 * a^2$$

$$a^2 \rightarrow a * a$$

Exponentiation by squaring

- Compute a^n where n is a nonnegative integer

For even values of n

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

For odd values of n

$$a^n = (a^{(n-1)/2})^2 a$$

Exponentiation by squaring

- Compute a^n where n is a nonnegative integer

```
power(a, n):  
1.     if (n = 1)  
2.         return a  
3.     if (n % 2 = 0)  
4.         t = power(a, n/2)  
5.         return t*t  
6.     else:  
7.         t = power(a, (n - 1) / 2)  
8.         return a * t*t
```

Efficiency = ???

Efficiency of exp-by-sqr

$$a^{38} \rightarrow a^{19} * a^{19}$$

$$a^{19} \rightarrow a * a^9 * a^9$$

$$a^9 \rightarrow a * a^4 * a^4$$

$$a^4 \rightarrow a^2 * a^2$$

$$a^2 \rightarrow a * a$$

How many
steps?

$\log_2 n$

How many
operations per step?

1 or 2
worst case 2

$O(\log n)$

Fake coin problem

Example of “decrease by factor of 2”

i.e. solve problem of size $n/2$

(Bonus: alternate solution that is “decrease by factor of 3”)

Fake coin problem

- A mischievous banker gives you n identical-looking coins, but tells you one is a fake (it is made from a lighter metal). Luckily, you have a balance scale, and can compare any two sets of coins.
- Design an efficient Decrease by a Constant Factor algorithm that finds the fake coin.



Key observation:

- Divide the pile in half
- Half on each side of balance
- Lighter half has the fake



- We eliminate HALF the coins in one step

Picky details

- What if n is odd?
 - Set aside one coin, then divide and weigh
 - Lighter pile \rightarrow fake coin is there
 - Equal piles \rightarrow fake coin is the extra (bonus!)
- Repeat the procedure until down to only 2 (or 3) coins

Fake coin problem

- Assume that $n=17$. How many times will you need to use the scale? Give two answers, one for the best case and one for the worst case.
- Best case: 1 weight comparison is needed.
- Worst case: 4 weight comparisons are needed.

$$\lfloor \log_2 n \rfloor$$

Fake coin problem

START:

- if $n=1$ the coin is fake

- else

 - if n is odd

 - remove first coin c_0 and set aside

 - else

 - divide remaining coins into two piles c_1 and c_2 , each with $\lfloor n/2 \rfloor$ coins

 - weigh the two piles

 - if they weigh the same

 - c_0 is the fake

 - else

 - discard the heavier pile and set $n = \lfloor n/2 \rfloor$

 - goto START

Fake coin problem

- This solution is $O(\log_2 n)$
 - It involves dividing the problem in half every time
- There is a better solution
 - Runs in $O(\log_3 n)$
 - Divide into 3 piles, weigh two of them
 - If different
 - Continue with the lighter pile (1/3 of the original)
 - If same
 - Continue with the unweighed pile (1/3 of the original)

Something to ponder

- The 3-pile solution is better by actual running time
- $\log_3(n)$ is less than $\log_2(n)$
- But they are both $O(\log n)$
- So how much “better” is the 3-pile solution?
 - What is $\log_3(100)$ vs. $\log_2(100)$?
 - How about $\log_3(1000000)$ vs. $\log_2(1000000)$?
- P.S. the 3-pile solution has a slightly trickier “base case”

Variable size decrease

Amount of decrease changes at each iteration

Euclid's algorithm

- Problem: Find $\text{gcd}(m,n)$, the greatest common divisor of two nonnegative numbers
- Examples: $\text{gcd}(8,6) = 2$
 - Divisors of 6 are 1, 2, 3, 6
 - Divisors of 8 are 1, 2, 4, 8

Euclid's algorithm

- Example $\text{gcd}(60,24) = ?$

- Euclid's algorithm:

$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

- until the second number becomes 0
- $\text{gcd}(60,24) = \text{gcd}(24,12) = \text{gcd}(12,0) = 12$

Euclid's algorithm (recursive) pseudocode

GCD (m, n)

1. if $((m \% n) = 0)$
2. return n
3. else
4. return GCD(n, $m \% n$)

Euclid's algorithm (iterative) pseudocode

1. GCD(m,n)
2. while $n \neq 0$ do
3. $r \leftarrow m \bmod n$
4. $m \leftarrow n$
5. $n \leftarrow r$
6. return m

Practice problems

- And for some *ON-TOPIC* problems (decrease-and-conquer):
 - Chapter 4.1, page 137, questions 7, 10
 - Chapter 4.4, page 156, question 3, 9