Decrease and Conquer

(Chapter 4)

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

Decrease-and-Conquer

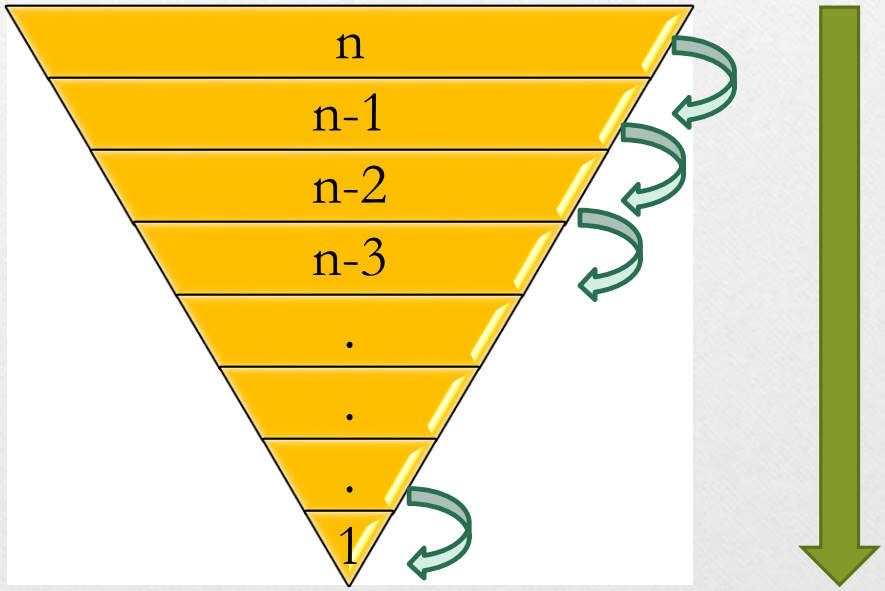
1. Reduce problem instance to smaller instance of the same problem and solve smaller instance
2. Extend solution of smaller instance to obtain solution to original instance

Can be implemented:

- top-down (Recursive)
- bottom-up (Iterative)

Decrease-and-Conquer

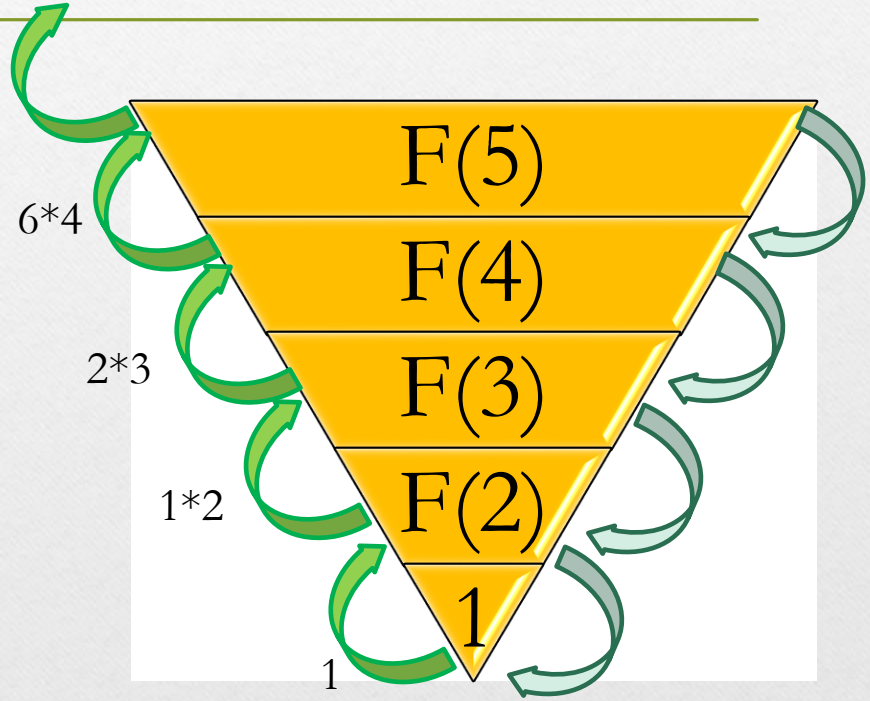
- Can be implemented:



top-down (Recursive)

Example: top-down (Recursive)

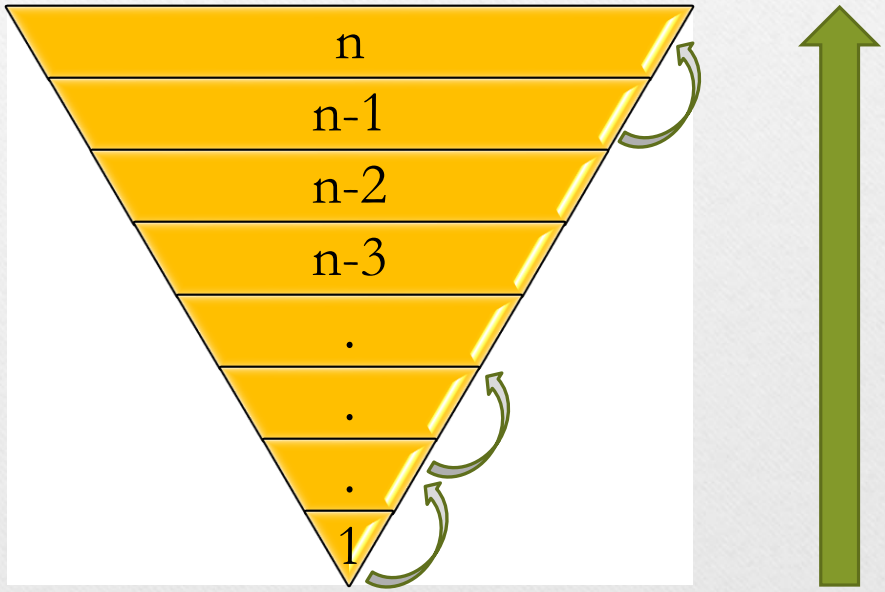
```
Factorial (n)
        if n = 1 then
    return 1
        else
    return Factorial(n − 1) × n
```



Factorial (5)= ?

Decrease-and-Conquer

- Can be implemented:

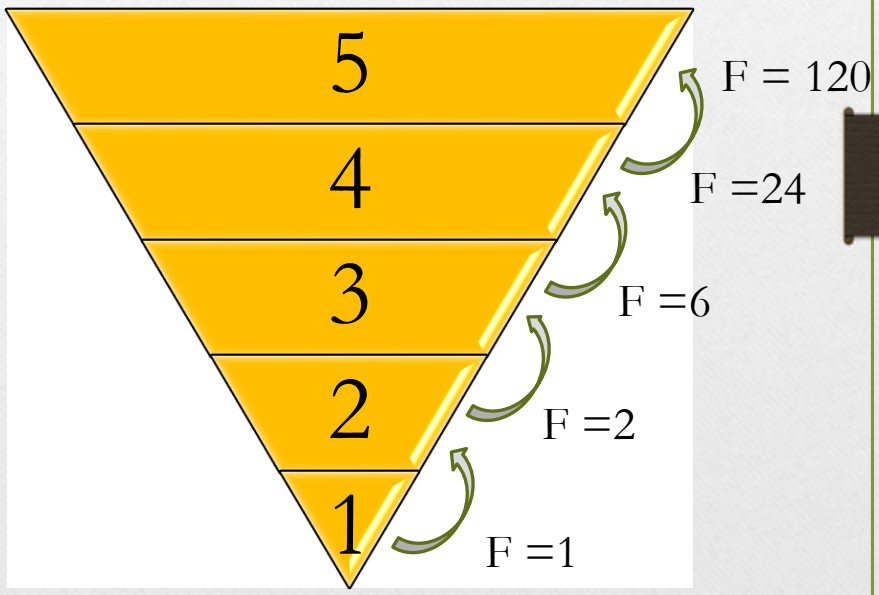


bottom-up

Example: bottom-up (Iterative)

```
Factorial (n)
        F ← 1
        for i← 1 to i← n
            F ← F *i
      return F
```

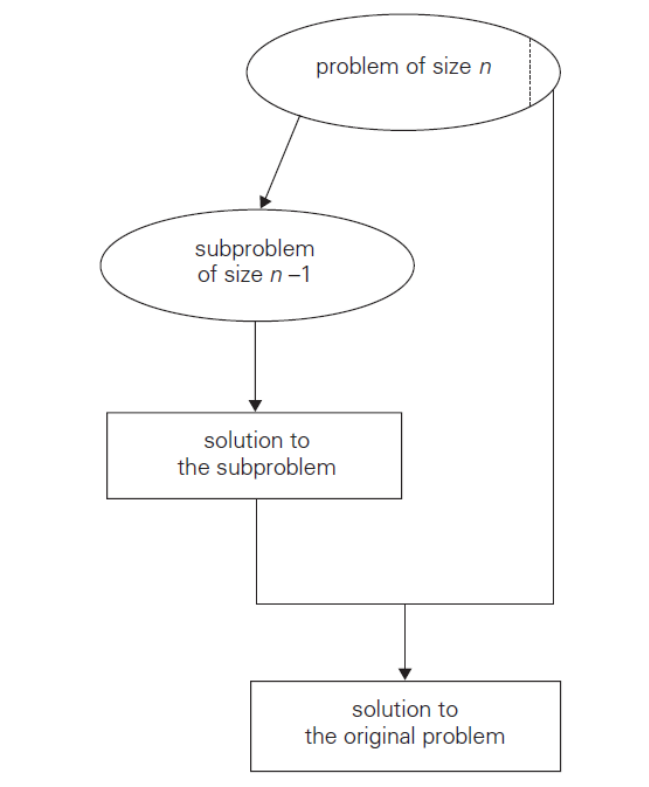
Factorial (5)= ?



Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

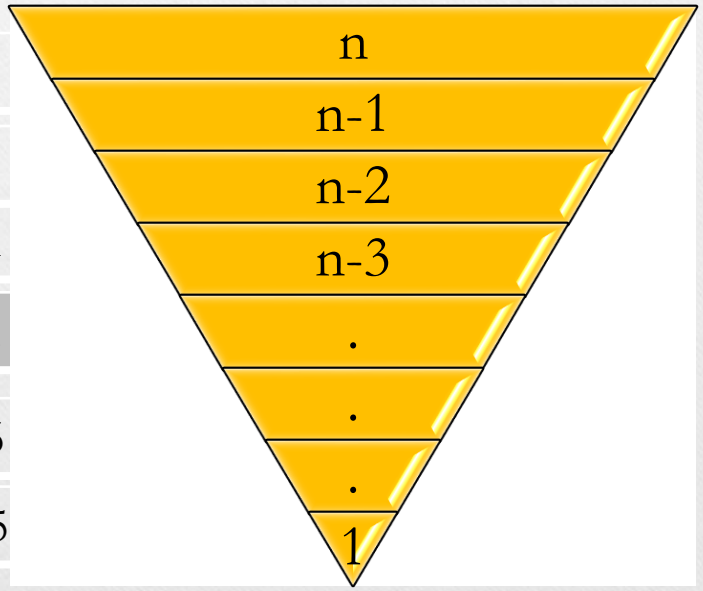
Decrease by One



Decrease by One (Insertion Sort)

```
1. Loops(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
```

5	**2**	4	6	1
2	5	**4**	6	1
2	4	5	**6**	1
2	4	5	6	**1**
1	2	4	5	6
1	2	3	4	5

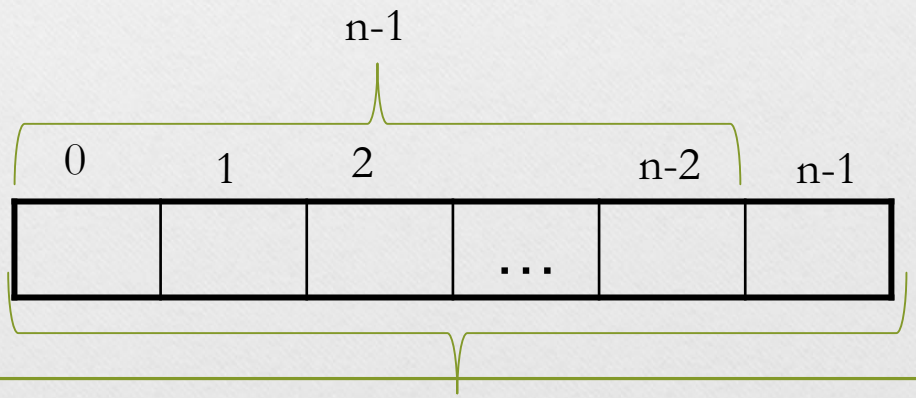


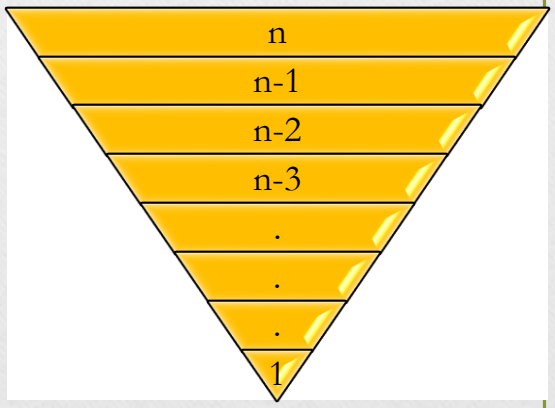
bottom-up

Decrease by One (Insertion Sort)

Insertion sort (A[0..n-1])

- Sort A[0..n-2]
- Insert element A[n-1] in its proper place among the sorted list A[0..n-2]



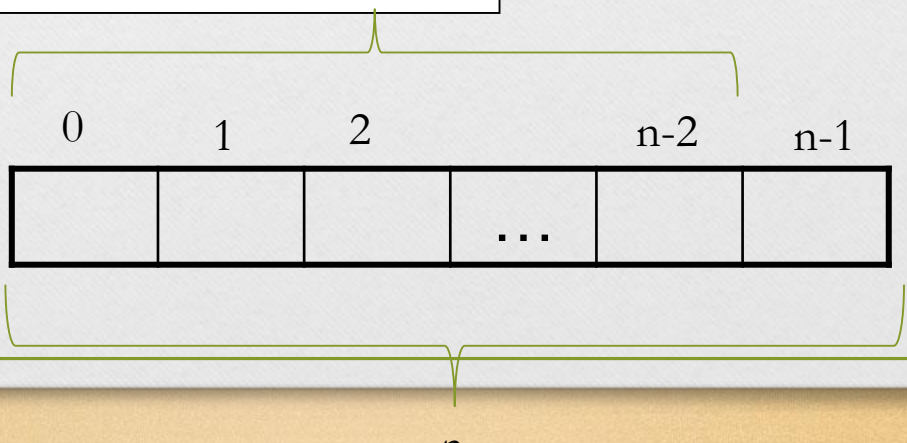


top-down (Recursive)

Decrease by One (Insertion Sort)

1.1 Insertion sort (recursive)

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



Decrease by One (Insertion Sort)

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

5	2	4	6	1	3	
5						
5	2					
2	5	4				
2	4	5	6			
2	4	5	6	1		
1	2	4	5	6	3	
1	2	3	4	5	6	

Recursive decrease

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

Decrease by One (Generating permutations)

What is a permutation?

For A,B,C:

ABC, ACB, BAC, BCA, CAB, CBA

Decrease by One (Generating permutations)

To find all permutations of n objects:

- Find all permutations of n-1 of those objects
- Insert the remaining object into all possible positions of each permutation of n-1 objects

Decrease by One (Generating permutations)

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

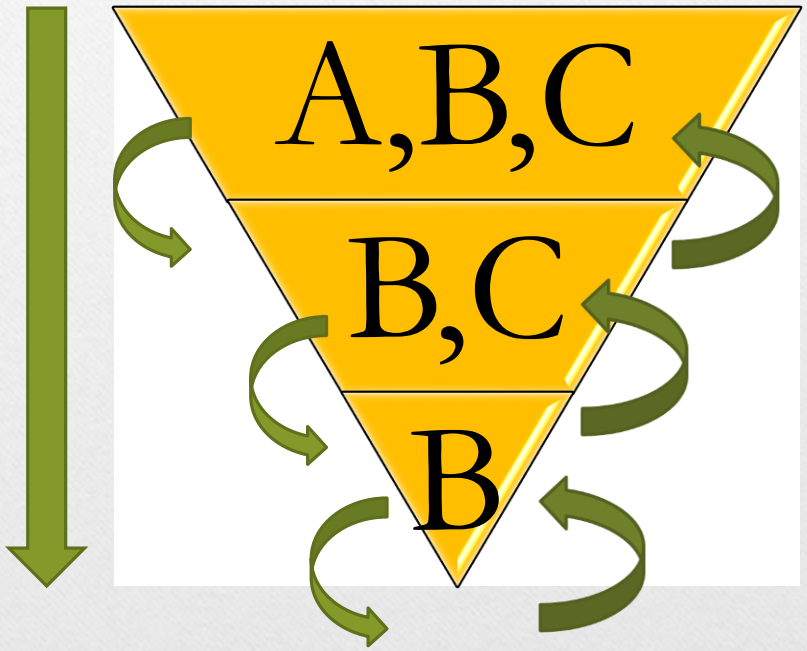
    B C and C B

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

    ABC BAC BCA and ACB CAB CBA

Decrease by One (Generating permutations)

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



ACB CAB CBA ABC BAC BCA

CB BC

B

Let's Code it!

Decrease by One (Generating permutations)

generatePermutation (n elements a_1, a_2, .., a_n)

If n = 1

return a_1

If n>1

Permutations ← generatePermutation (n-1 of elements a_1, a_2, .., a_{n-1})

for each p in permutations

insert a_n before a_1

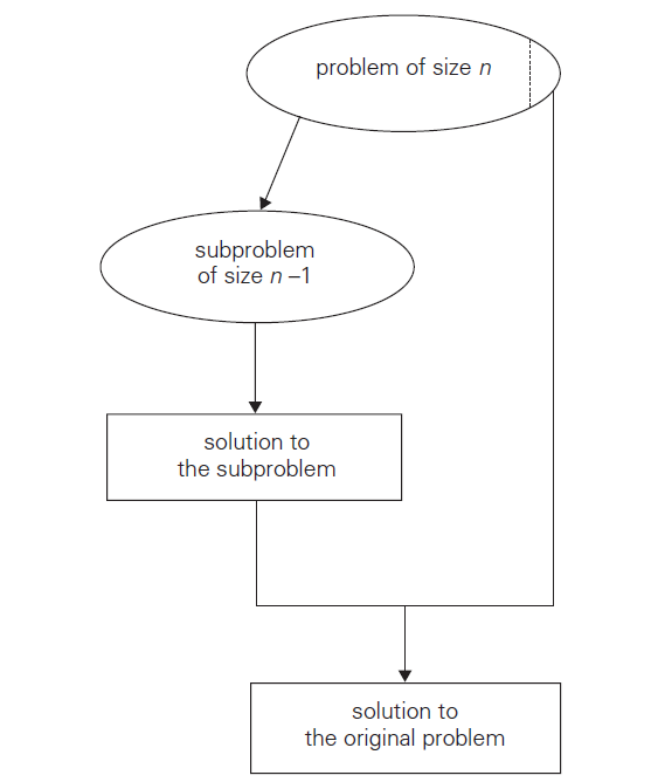
for i← 1 to i←n-1

insert a_n after a_i

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

Decrease by One



Decrease by One (Generating Subsets)

What is a subset?

For a,b,c:

{a, b ,c}, {a, b}, {b, c}, {a, c}, {a}, {b}, {c}, {}

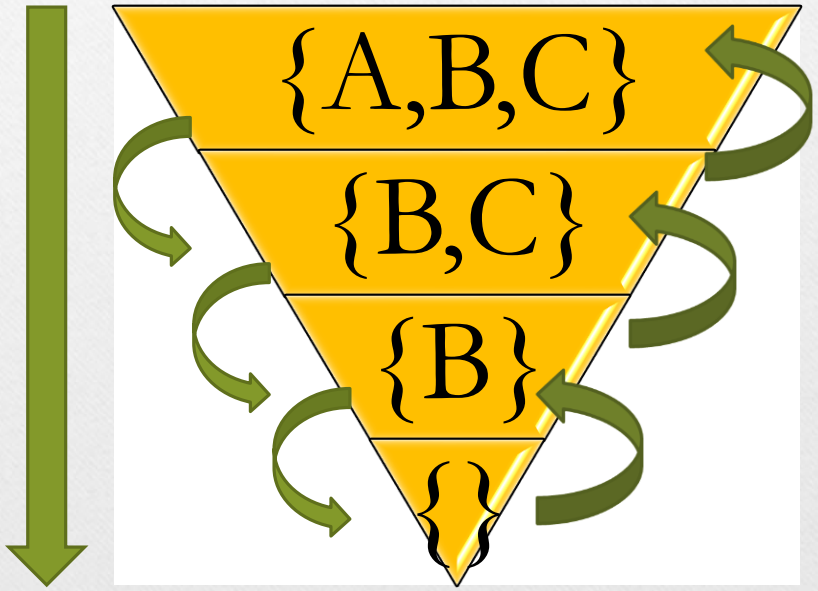
Decrease by One (Generating Subsets)

To find all subsets of `n` objects:

- Find all subsets of `n-1` of those objects
- For each subsets copy it and insert the remaining object to the subset

Decrease by One (Generating permutations)

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



{} {B} {C} {B C}
{A} {A B} {A C} {A B C}

{} {B} {C} {B C}

{} {B}

{}

Decrease by One (Generating permutations)

generateSubsets (n elements a_1, a_2, .., a_n)

If n=0

return empty set

If n>0

subsets ← generateSubsets (n-1 elements a_1, a_2, .., a_{n-1})

for each subset s in subsets

clone s

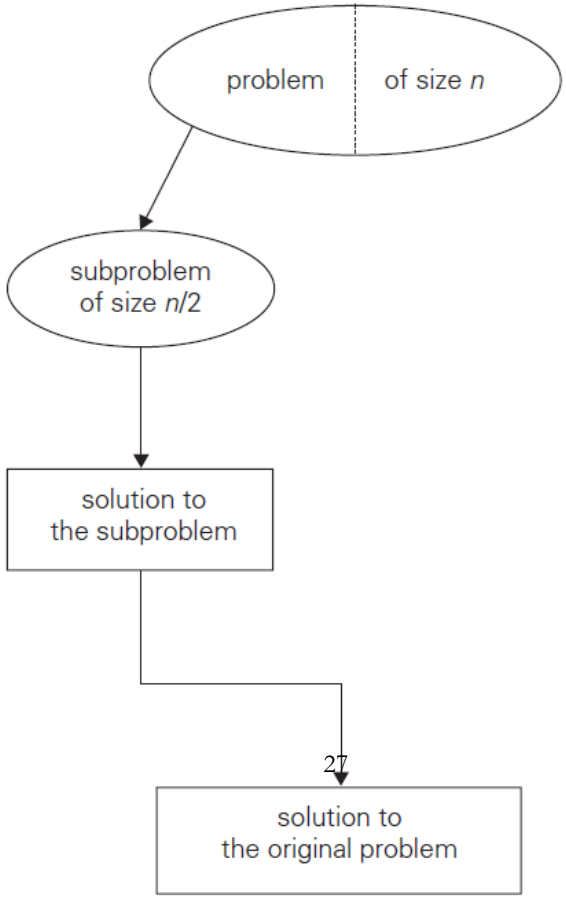
insert a_n to the s

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

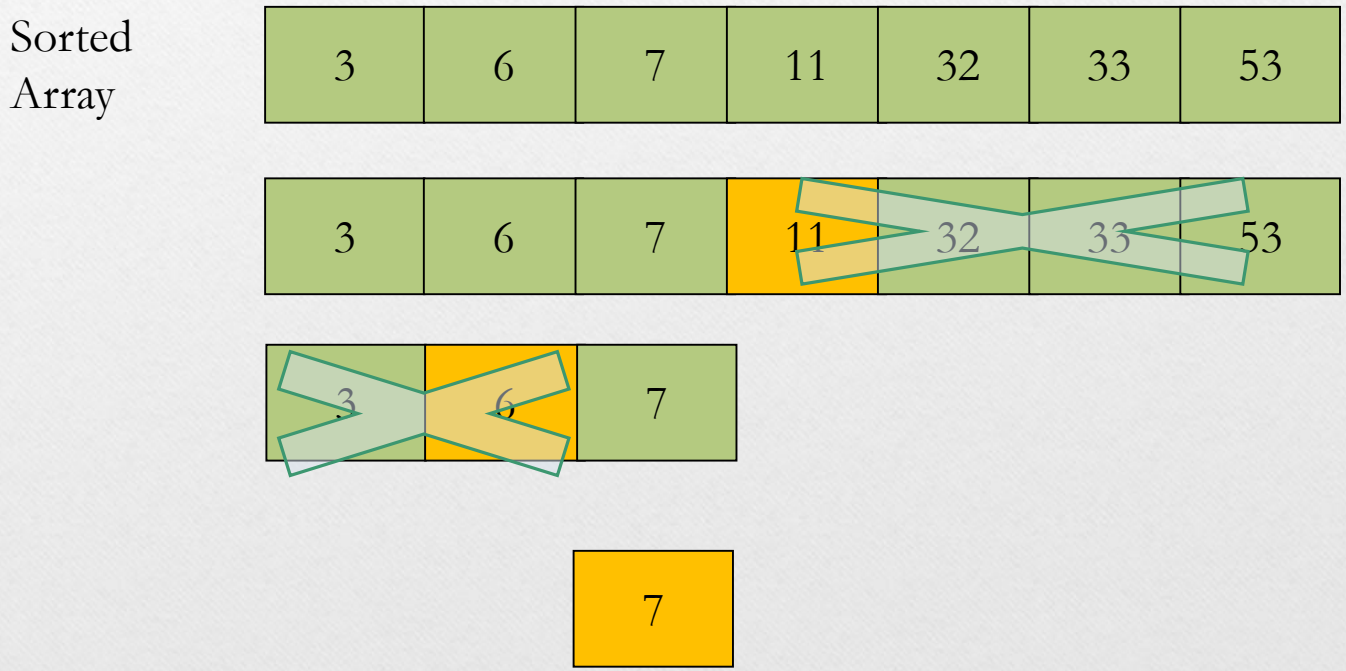
Decrease by Half

- Make the problem smaller by some constant factor.
- Typically the constant factor is *two*, ie, we divide the problem in half.



Decrease by Half (Binary Search)

- 2.1 Binary search, key =7



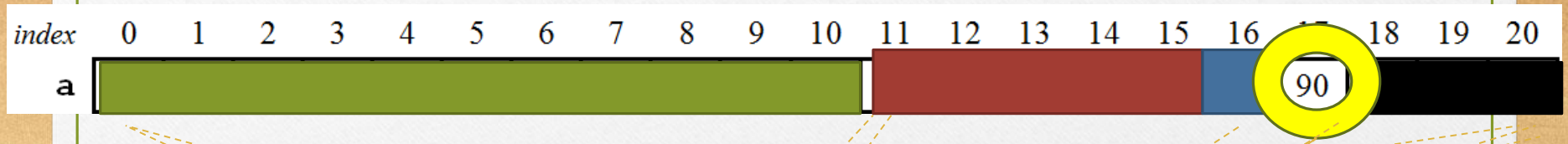
Decrease by Half (Binary Search)

- Binary Search *works by dividing the sorted array (ie: the solution space) in half each time,* and searching in the half where the target should exist
- *In other words, we throw away 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

2. Decrease by half

2.1 Binary search (Recursive)

Example: for k=90



Call trace:
binarySearch(a[], k, start, end)

1. binarySearch(a, 90, 0, 20)
1.1 binarySearch(a, 90, 11, 20)
1.1.1 binarySearch(a, 90, 11, 17)
1.1.1.1 binarySearch(a, 90, 16,17)
1.1.1.1.1 binarySearch(a, 90, 17, 17)
***target found, returns*

pseudocode

- You try! Attempt to write pseudocode implement:
  - binarySearch(a[], k, start, end)

Decrease by Half (Binary Search)

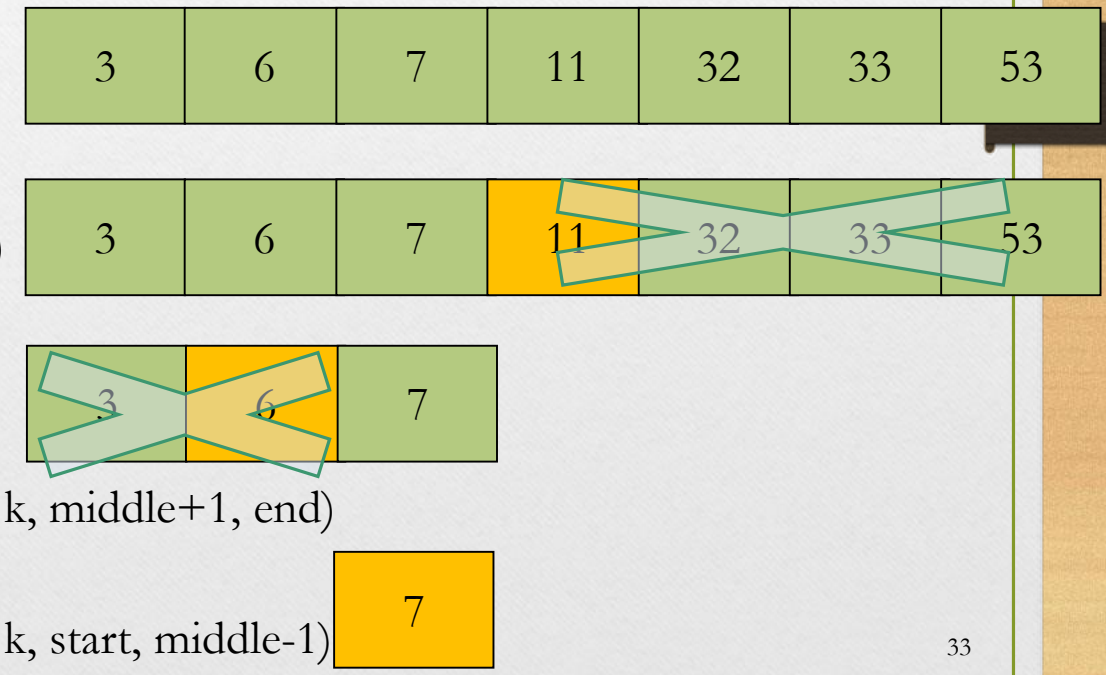
- 2.1 Binary search

```
binarySearch(a[], k, start, end)
    if end < start
            return not found
    middle ← floor((start+end)/2)
    if  k = a[middle]
            return found
    if k > a[middle]
            return binarySearch(a[], k, middle+1, end)
    else if k < a[middle]
            return binarySearch(a[], k, start, middle-1)
```

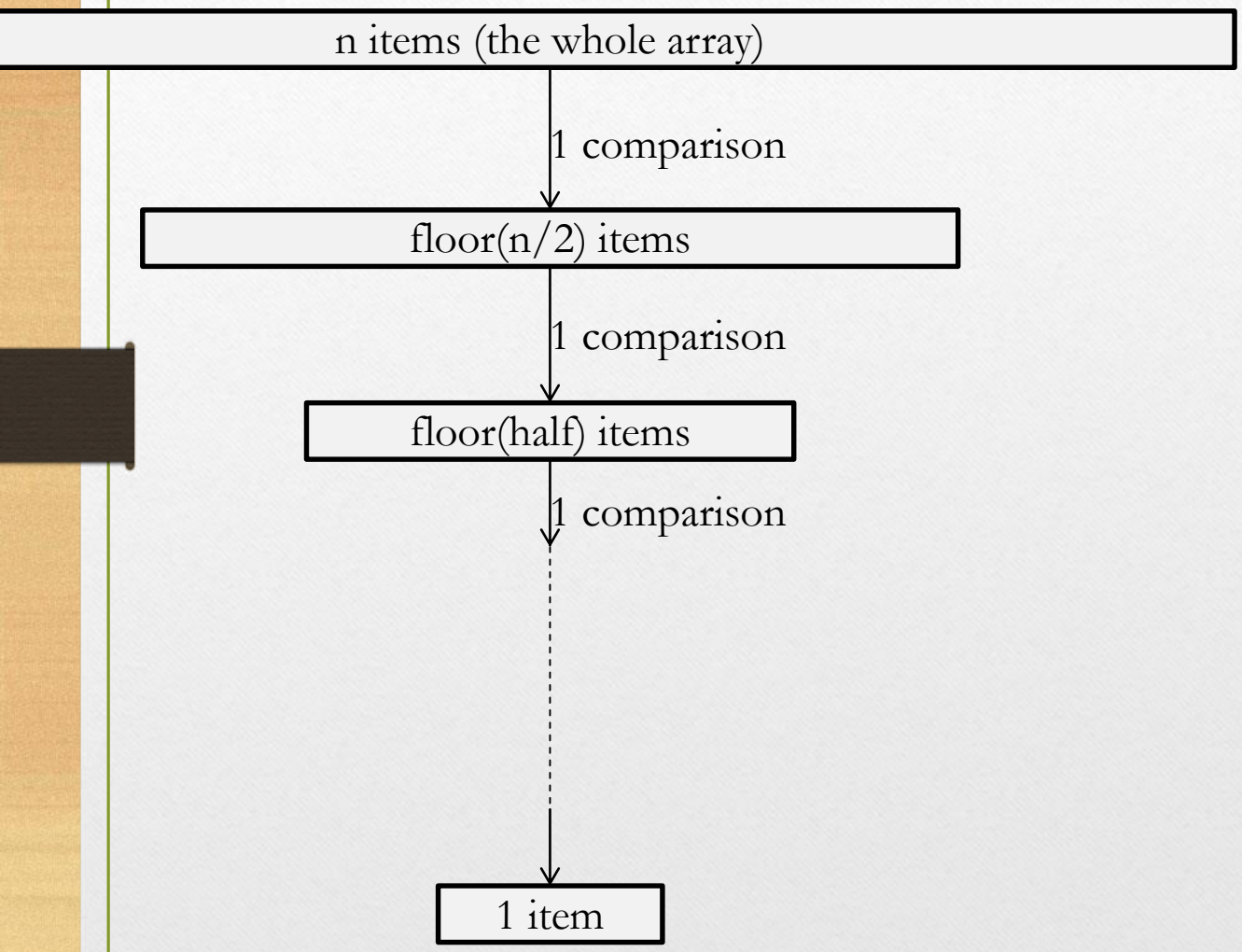
Decrease by Half (Binary Search)

- 2.1 Binary search, key =7

```
binarySearch(a[], k, start, end)
    if end < start
            return not found
    middle ← floor((start+end)/2)
    if  k = a[middle]
            return found
    if k > a[middle]
            return binarySearch(a[], k, middle+1, end)
    else if k < a[middle]
            return binarySearch(a[], k, start, middle-1)
```



Binary Search Efficiency



How to think of it:
Imagine n
is 64.

What power
of 2 equals
64? (ans 6)

In general:
What power of 2 equals
n?

$log_2 n = e$
Means:
$n = 2^e$

So:
We roughly want $log_2 n$

Actually $log_2 n + 1$
(final comparison)

Binary Search Efficiency

- Time efficiency
  - Worst-case efficiency…
    - $C_w(n) = \log_2(n+1)$
    - So binary search is O(log n)

  This is VERY fast: e.g., $C_w(1000000) = 20$

- Optimal for searching a sorted array
- Limitations: must be a sorted array

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

Decrease by half

(Fake Coin Problem)

Assume that you have n identical looking coins, but one is a fake (weighs less). You also 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.

Fake Coin Problem (Example)

Question 1: Assume that n=2. How many times will you need to weigh the coins?

One weight comparisons are needed.

Fake Coin Problem (Example)

Question 2: Assume that n=4. How many times will you need to weigh the coins? Give two answers, one for the best case and one for the worst case.

Worst case: 2 weight comparisons are needed.

Best base: 2 weight comparisons are needed.

Fake Coin Problem (Example)

Question 3: Assume that n=8. How many times will you need to weigh the coins? Give two answers, one for the best case and one for the worst case.

Worst case: 3 weight comparisons are needed.

Best base: 3 weight comparisons are needed.

$\log_2 n$

Fake Coin Problem (Example)

Question 3: Assume that n=9. How many times will you need to weigh the coins? Give two answers, one for the best case and one for the worst case.

Worst case: 3 weight comparisons are needed.

Best base: 1 weight comparisons are needed.

$\lceil \log_2 n \rceil$

Fake Coin Problem (Algorithm)

Explain, in plain English, how you would solve this problem.

Divide the coins into two equal piles. If n is odd, set one coin aside first. Compare the piles (ie: put one pile on each side of the balance scale).

If the piles weigh the same, the coin that was put aside is the fake, otherwise the fake is in the pile that has lesser weight.

Discard the heavier pile. Split the remaining pile in half and repeat the above procedure until there are only two coins, or, the lighter coin has been found.

If there are only two coins left, the lighter of the two is the fake.

Fake Coin Problem (cont …)

Question 4: Write the pseudocode for your solution to this problem.

```
fakeCoin(n coins):
    if n=1 the coin is fake
    else
        if n is odd
            remove first coin c0 and set aside
        else
            divide remaining coins into two piles c1 and c2, each with ⌊n / 2⌋ coins
        weigh the two piles
         if they weigh the same
            c0 is the fake
         else
            discard the heavier pile and set n = ⌊n / 2⌋
         fakeCoin(remaining pile)
```

Fake Coin Problem

- This solution is $O(\log_2 n)$
  - It involves dividing the problem in half every time
- Is there is a better solution?
  - Hint: yes, it runs in $O(\log_3 n)$
  - Divide into 3 piles, weigh two of them
  - If different
    - Continue with the lighter one (1/3 of the original)
  - If same
    - Continue with the unweighed pile (1/3 of the original)

Decrease and Conquer

1. Introduction to Decrease and Conquer
2. Decrease by One
   - Insertion sort
   - Generating permutations
   - Generating subsets
3. Decrease by half
   - Binary search
   - Fake Coin Problem

Try it/ Homework

1. Chapter 4.1, page 137, questions 7,10
2. Chapter 4.4, page 156, question 3, 9

Notes for Lab Two

The sort algorithm for technique two can use Array.sort (java.util.Arrays' solution)

Quiz Time

Will be 20 minutes

Please be a silent as possible