Decrease and Conquer

(Chapter 4)

Decrease-and-Conquer

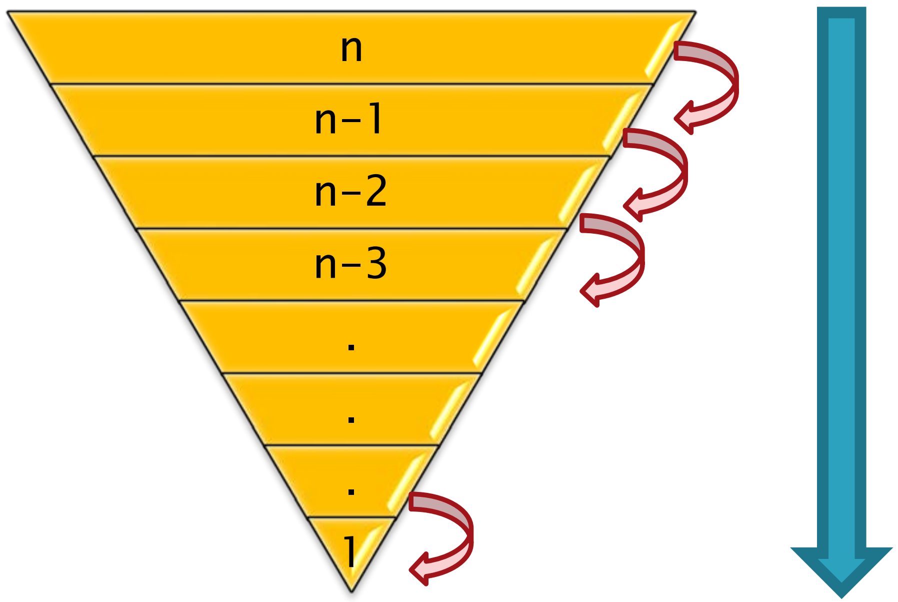
1. Reduce problem instance to smaller instance of the same problem and solve smaller instance
2. Extend solution of smaller instance to obtain solution to original instance

Can be implemented:

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

Decrease-and-Conquer

- Can be implemented:

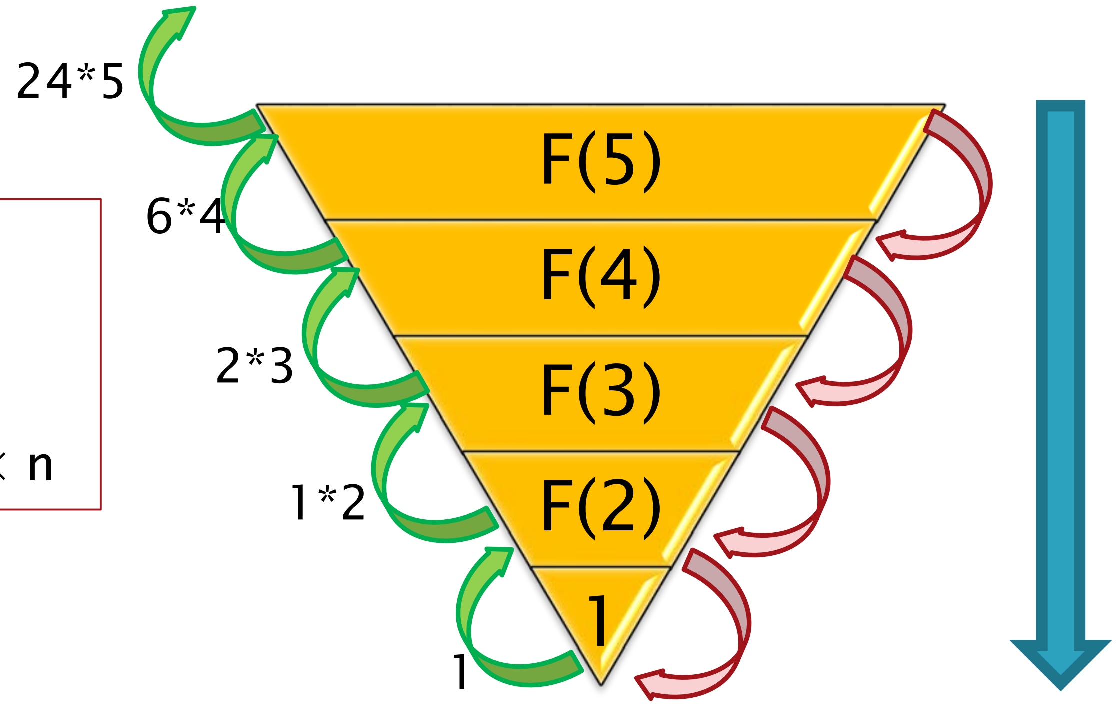


top-down (Recursive)

Example: top-down (Recursive)

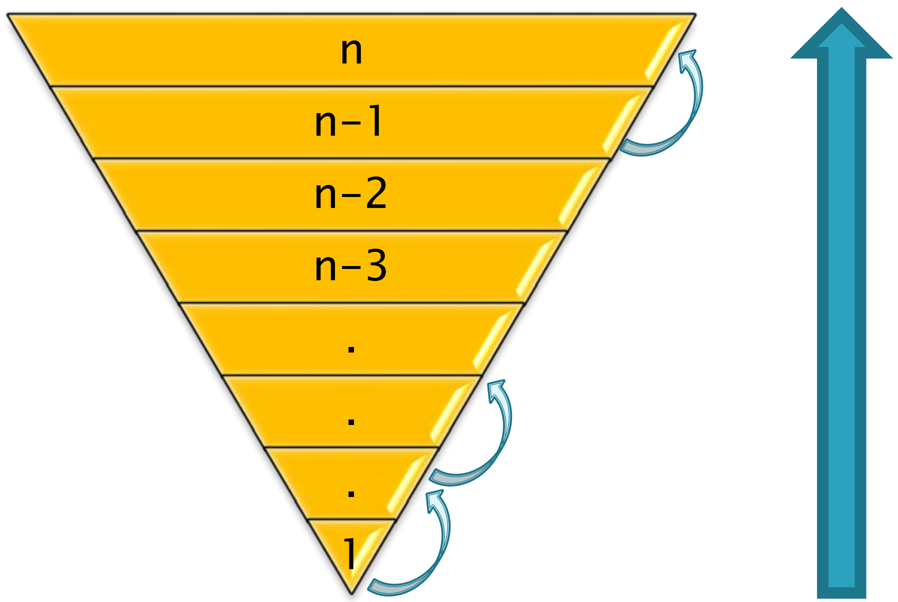
```
Factorial (n)
        if n = 1 then
        return 1
        else
        return Factorial(n − 1) × n
```

Factorial (5)= ?



Decrease-and-Conquer

- Can be implemented:

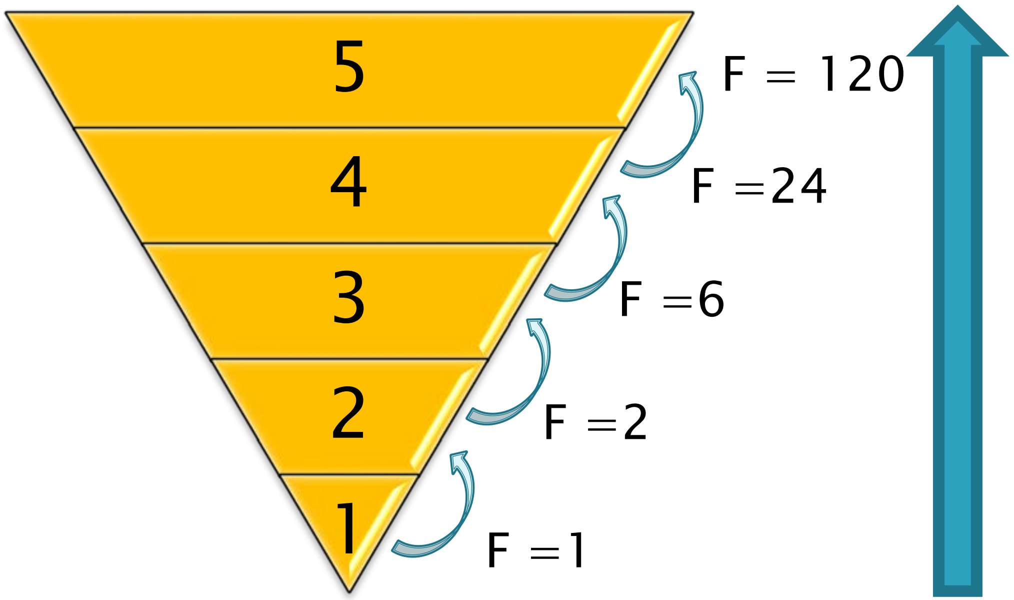


bottom-up

Example: bottom-up (Iterative)

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

Factorial (5)= ?

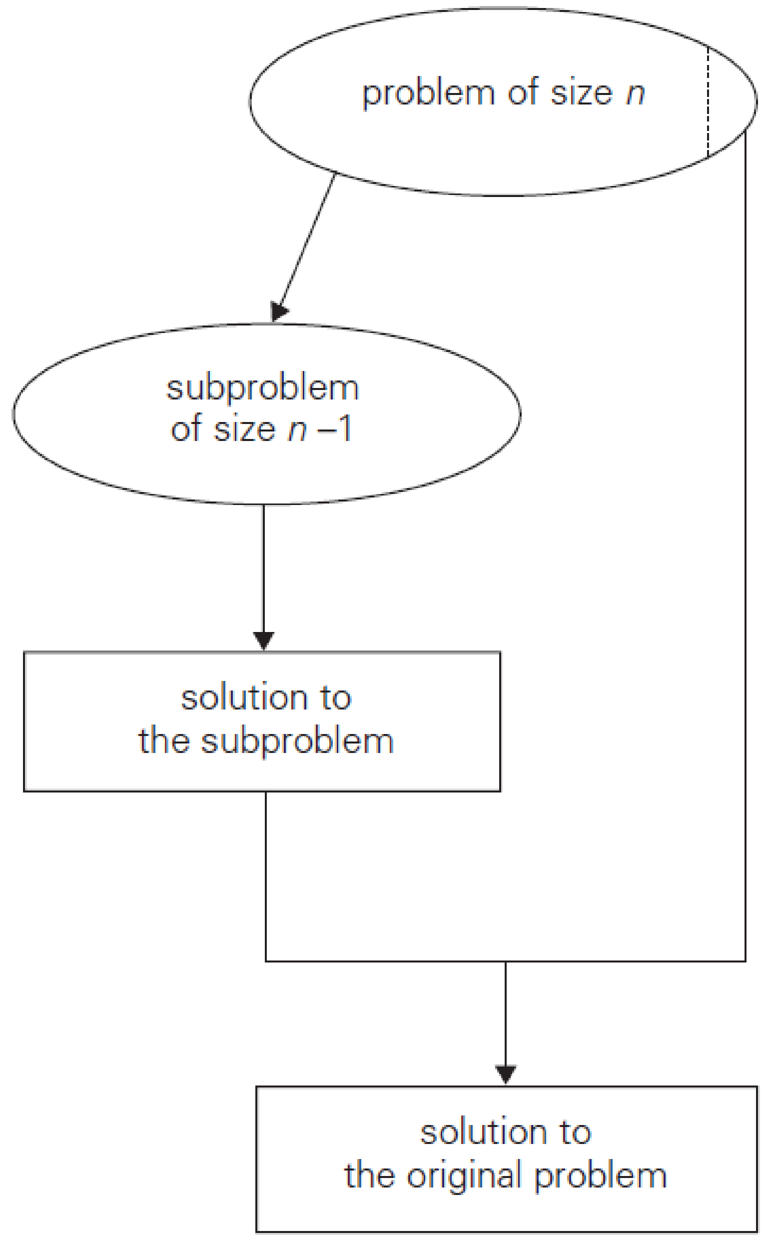


2 Types of Decrease and Conquer

1. Decrease by 1
   1.1 Insertion sort
   1.2 Generating permutations
   1.3 Generating subsets

2. Decrease by half
   2.1 Binary search
   2.2 Exponentiation by squaring
   2.3 Fake Coin Problem

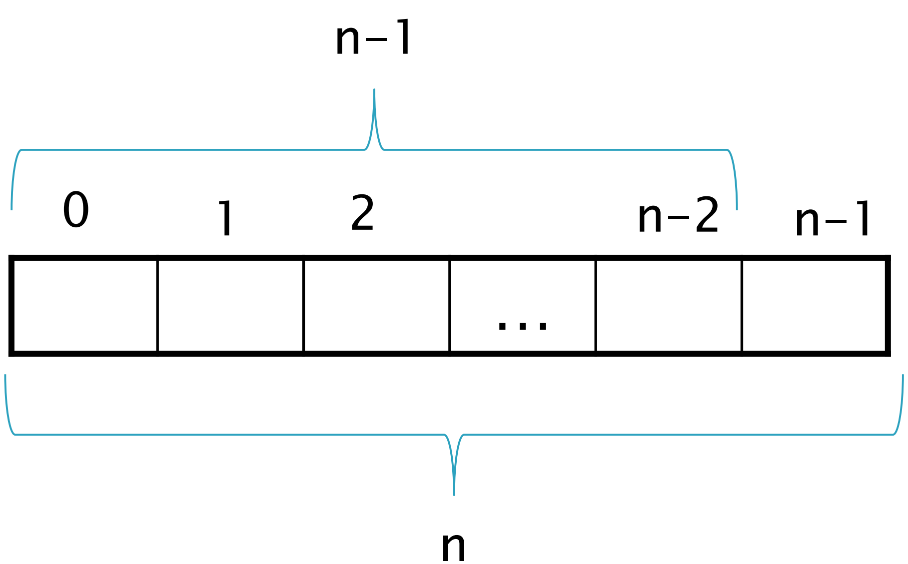
Decrease by 1

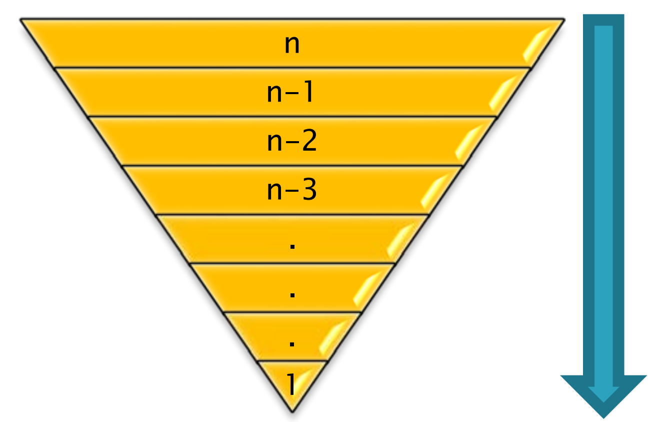


1.Decrease by 1

1.1Insertion sort (A[0..n−1])

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



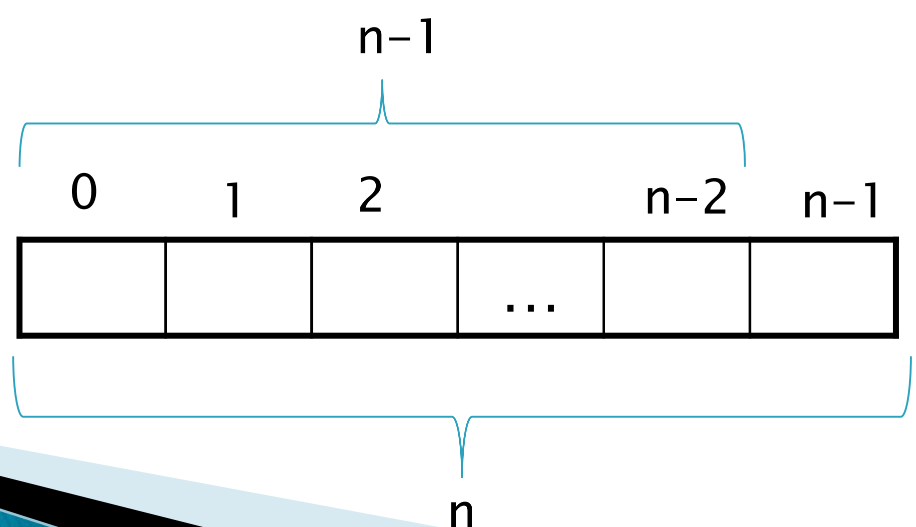


top−down (Recursive)

1.Decrease by 1

1.1 Insertion sort (recursive)

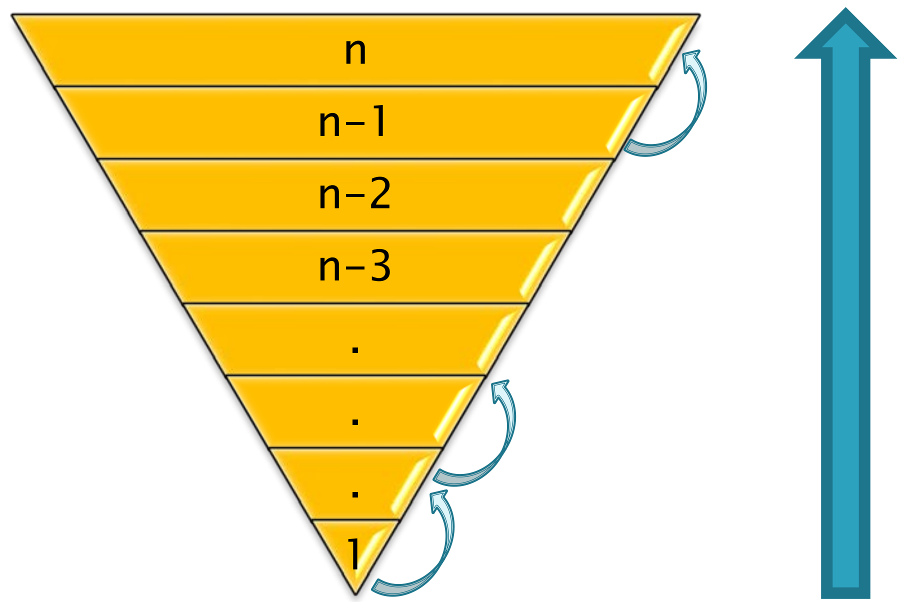
```
InsertionSort(A ,n)
if n > 1
      InsertionSort(A,n-1)
      key ← A[n-1]
      i = n-2
      while i ≥ 0 and A[i] > key
            A[i+1] ← A[i]
            i ← i - 1
      A[i + 1] ← key
```



1.Decrease by 1

1.1 Insertion sort (Iterative)

```
InsertionSort(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```



bottom-up

2 Types of Decrease and Conquer

1. Decrease by 1
   1.1 Insertion sort
   1.2 Generating permutations
   1.3 Generating subsets

2. Decrease by half
   2.1 Binary search
   2.2 Exponentiation by squaring
   2.3 Fake Coin Problem

1. Decrease by 1

1.2 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

1.2 Generating permutations

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

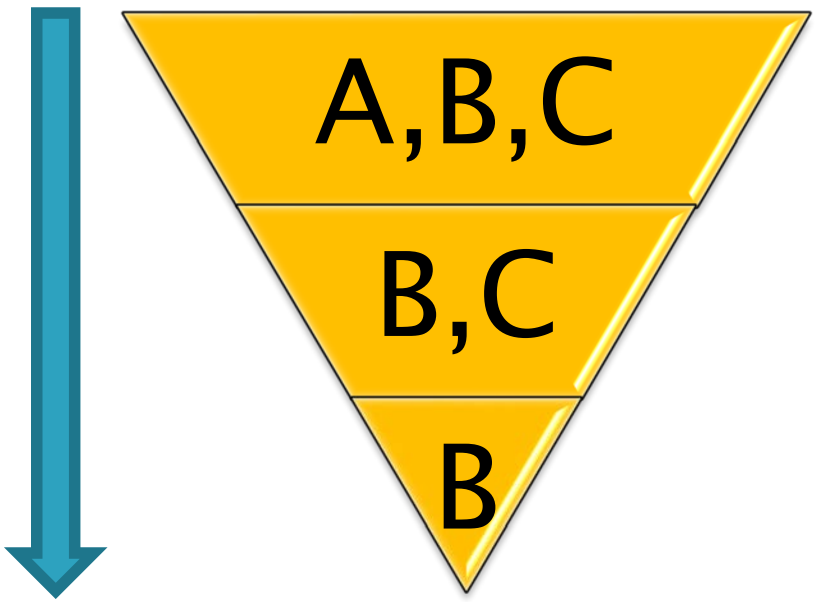
    B C and C B

  - 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

1.2 Generating permutations

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



ACB CAB CBA ABC
BAC BCA

CB BC

B

1.2 Generating permutations

- Pseudocode:

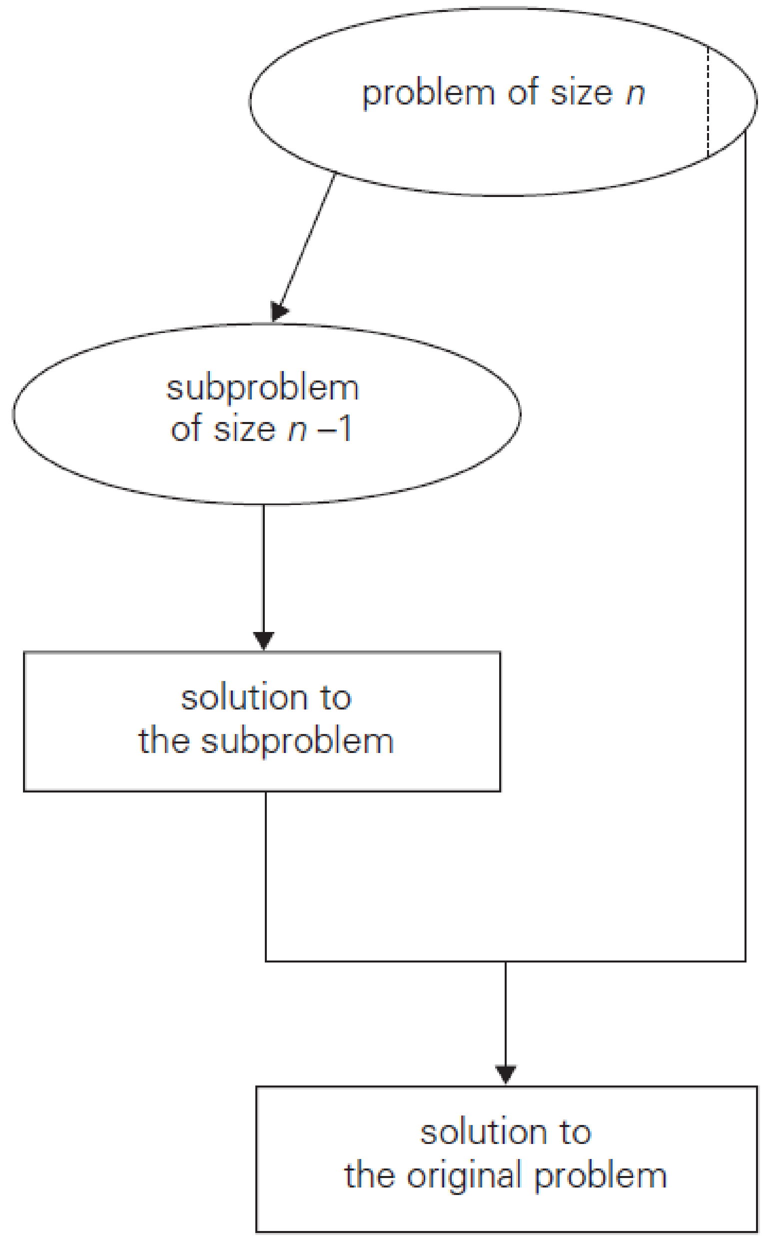
<discuss in class>

2 Types of Decrease and Conquer

1. Decrease by 1
   1.1 Insertion sort
   1.2 Generating permutations
   1.3 Generating subsets

2. Decrease by half
   2.1 Binary search
   2.2 Exponentiation by squaring
   2.3 Fake Coin Problem

Decrease by 1



1. Decrease by 1

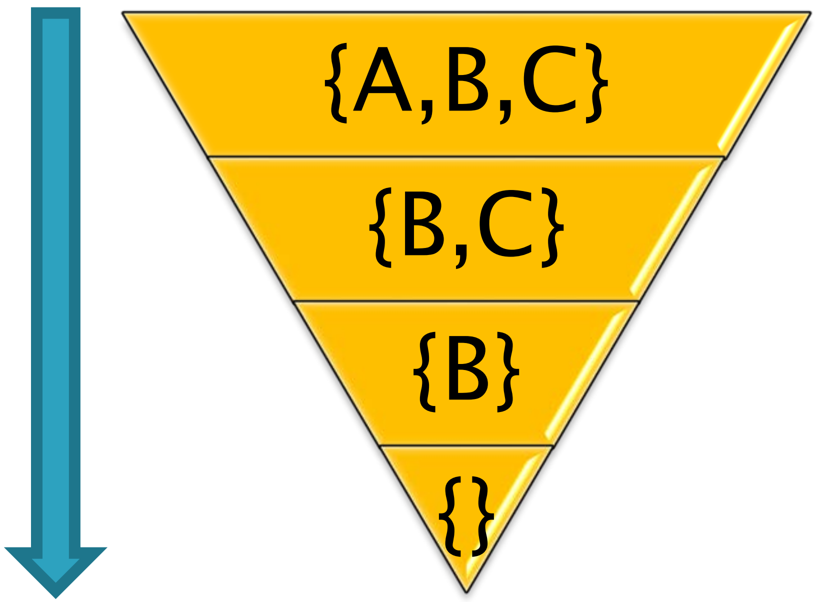
1.3 Generating subsets

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

1.3 Generating subsets

- 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}

{}

1.2 Generating subsets

- Pseudocode:

<discuss in class>

2 Types of Decrease and Conquer

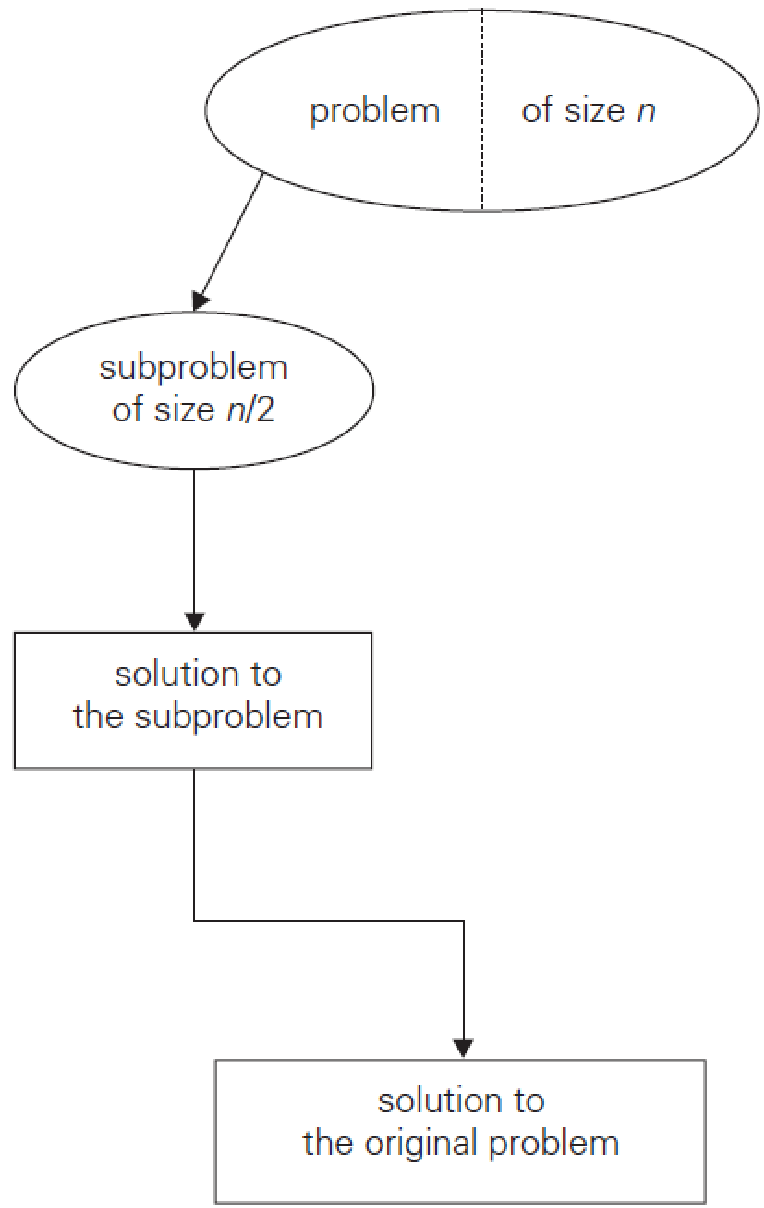
1. Decrease by 1
   - 1.1 Insertion sort
   - 1.2 Generating permutations
   - 1.3 Generating subsets

2. Decrease by half
   - 2.1 Binary search
   - 2.2 Exponentiation by squaring
   - 2.3 Fake Coin Problem

2. Decrease by half

- Make the problem smaller by some constant factor.

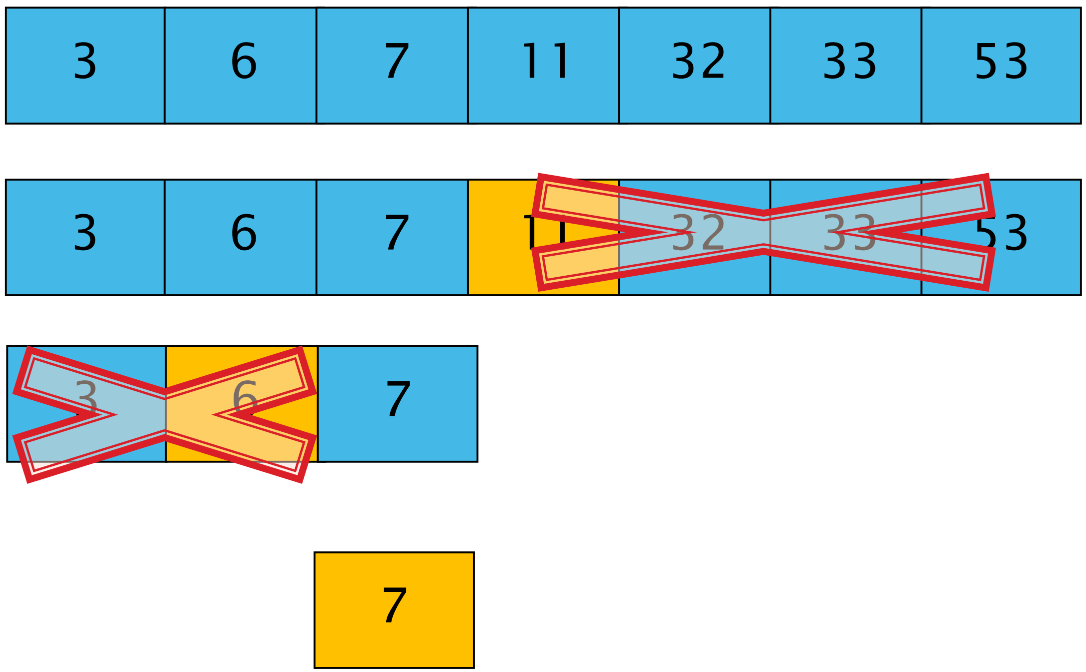
- Typically the constant factor is *two*, ie, we divide the problem in half.



2. Decrease by half

- 2.1 Binary search, key =7

Sorted Array



2. Decrease by half

2.1 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

pseudocode

- <discuss in class>

2. Decrease by half

2.1 Binary search (Recursive)

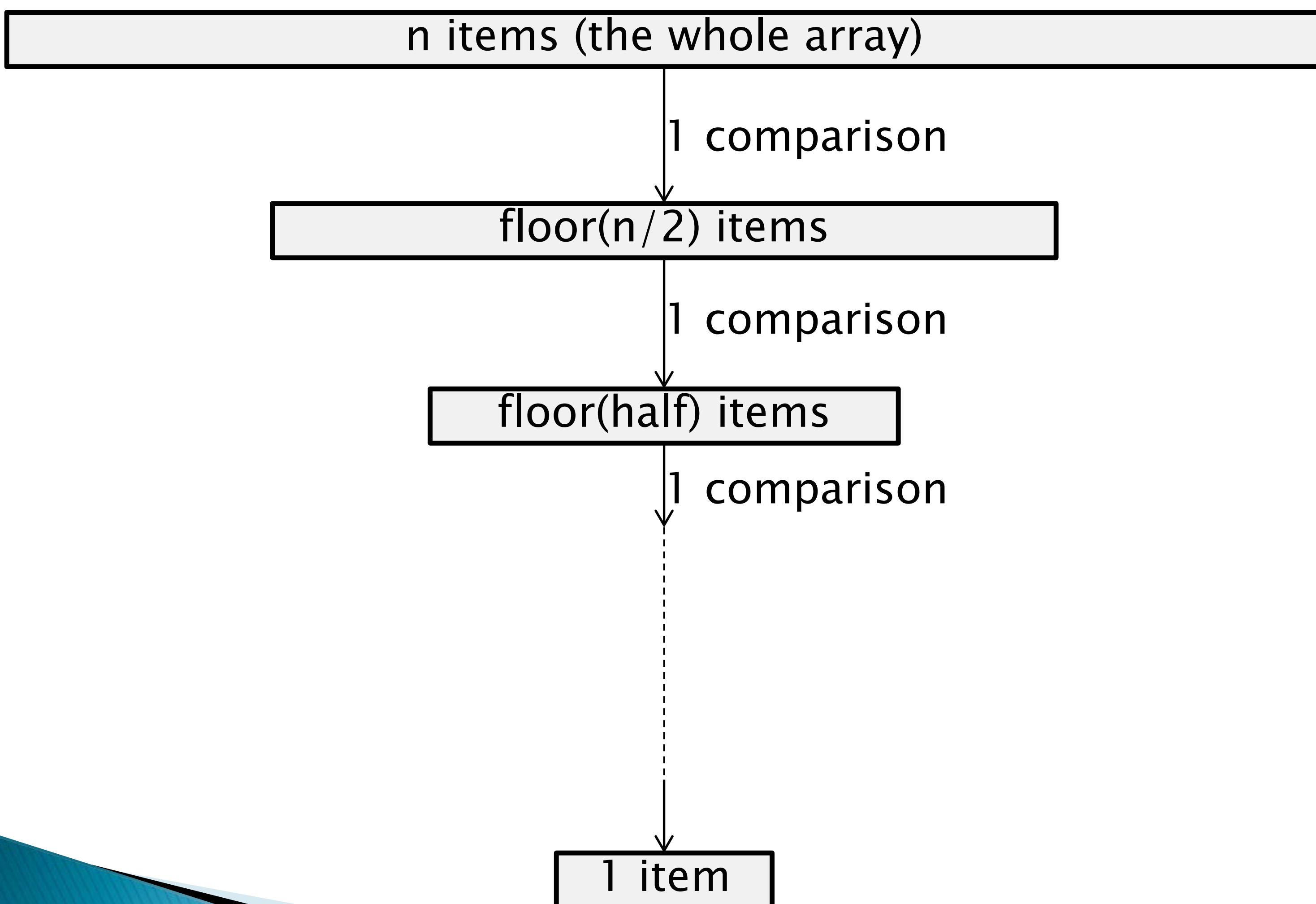
Example: for k=90

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

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



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?

High school math:

$log_2 n = e$
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

2. Decrease by half

2.1 Binary search (iterative)

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

2. Decrease by half

2.1 Binary search (iterative)

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

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

- Soln for k=81 (s=0, e= 20 initially)
  - iteration 1: s,e,m = 11,20,10
  - iteration 2: s,e,m = -,-,15 ** target found

- Soln 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 $\lceil \log_2 n \rceil + 1$)

2 Types of Decrease and Conquer

1. Decrease by 1
   - 1.1 Insertion sort
   - 1.2 Generating permutations
   - 1.3 Generating subsets
2. Decrease by half
   - 2.1 Binary search
   - 2.2 Exponentiation by squaring
   - 2.3 Fake Coin Problem

2. Decrease by half

2.2 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

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

2. Decrease by half

2.2 Exponentiation by Squaring Pseudocode:

<discuss in class>

2 Types of Decrease and Conquer

1. Decrease by 1
   1.1 Insertion sort
   1.2 Generating permutations
   1.3 Generating subsets

2. Decrease by half
   2.1 Binary search
   2.2 Exponentiation by squaring
   2.3 Fake Coin Problem

2. Decrease by half

2.3 Fake Coin Problem

Assume that you have n identical looking coins, but one is a fake (it is made from a lighter metal). 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 (solutions)

Question 1: 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.

<discuss in class>

Fake Coin Problem (cont …)

Question 2: Assume that n=17. How many times will you need to weigh the coins? Give two answers, one for the best case and one for the worst case.

<discuss in class>

Fake Coin Problem (solutions)

Question 3 :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.

```
START:
    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⌋
         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 one (1/3 of the original)
  - If same
    - Continue with the unweighed pile (1/3 of the original)

2 Types of Decrease and Conquer

1. Decrease by 1
   1.1 Insertion sort
   1.2 Generating permutations
   1.3 Generating subsets

2. Decrease by half
   2.1 Binary search
   2.2 Exponentiation by squaring
   2.3 Fake Coin Problem

QUIZ Announcement

- There will be a quiz in the lab next week.

- It will be 5 questions, on D2L
  - It will take 10-20 minutes
  - Followed by a lab activity

Try it/ Homework

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