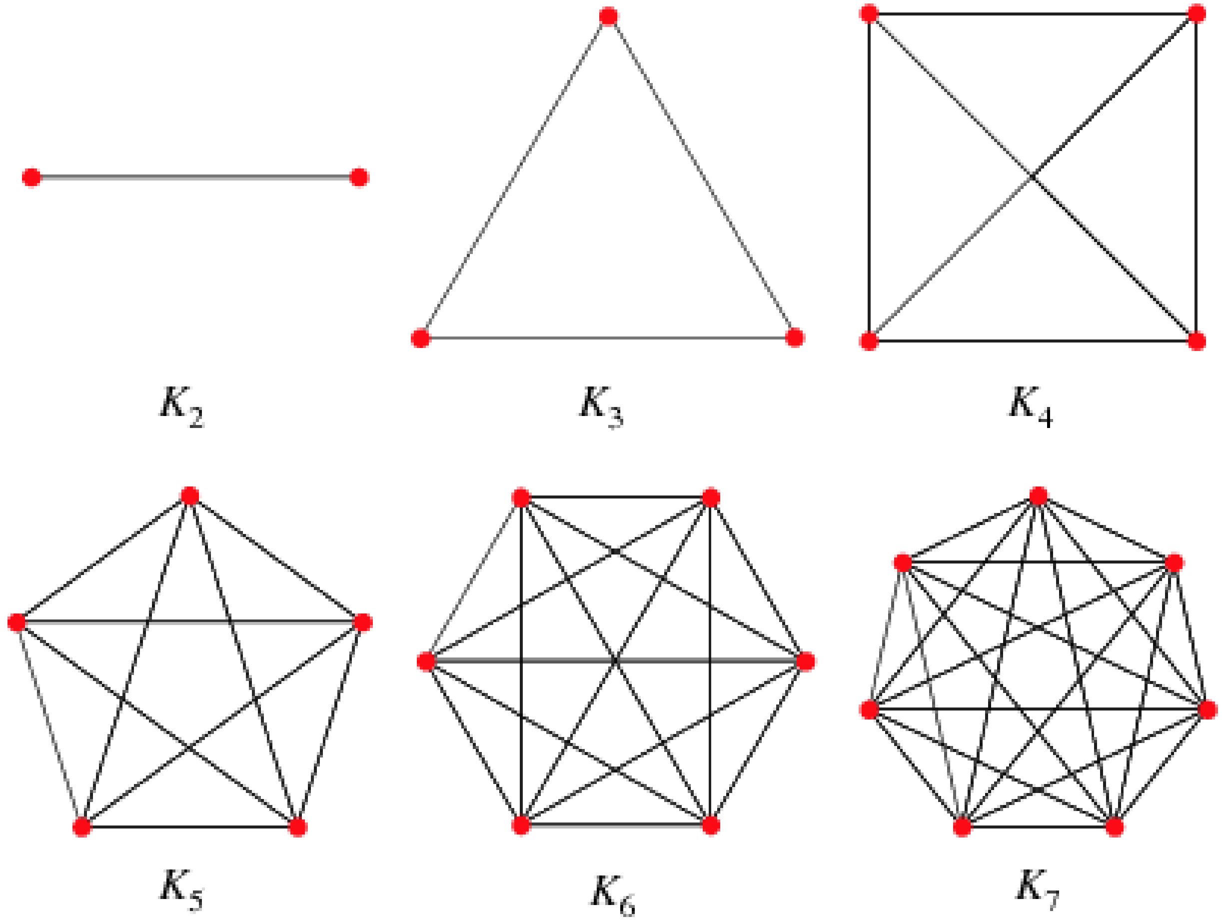
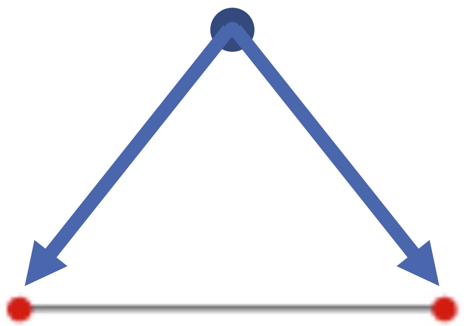
No attendance quiz today!

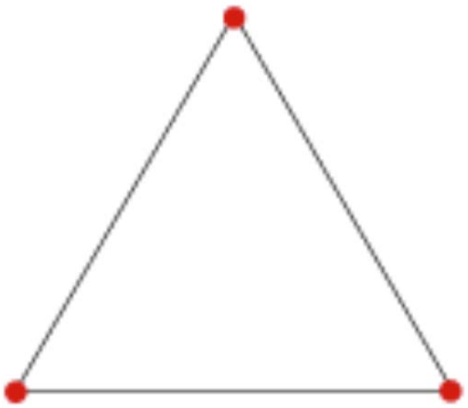
How many edges in a complete graph?



From K_2 to K_3

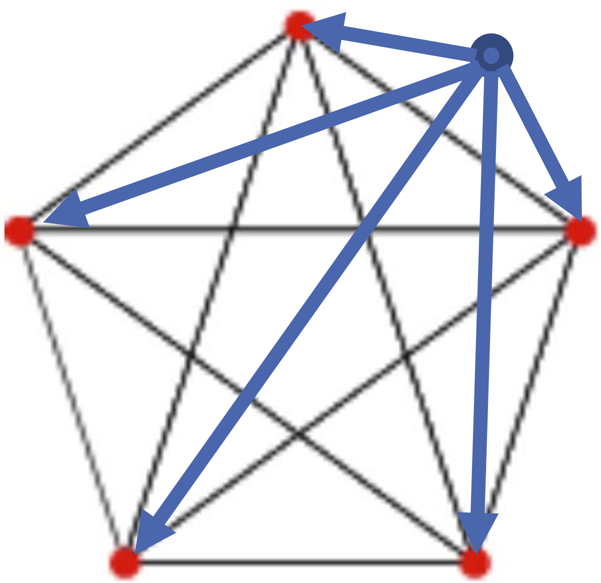
1. Add one vertex
2. Connect it to both of the other vertices

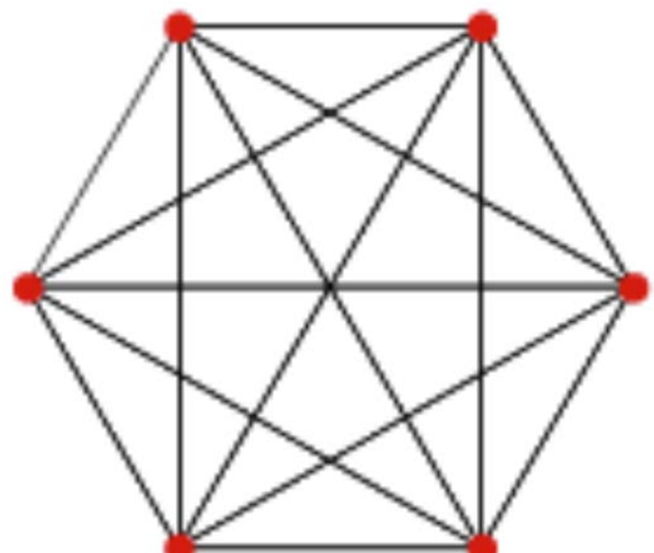




From K_5 to K_6

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





From K_n to K_{n+1}

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

- Number of edges in K_{n+1}
  = n + number of edges in K_n
- "Base case" K_2 has 1 edge

- num_edges(K_{37}) = 36 + num_edges(K_{36})

DECREASE-AND-CONQUER ALGORITHMS

(Chapter 4)

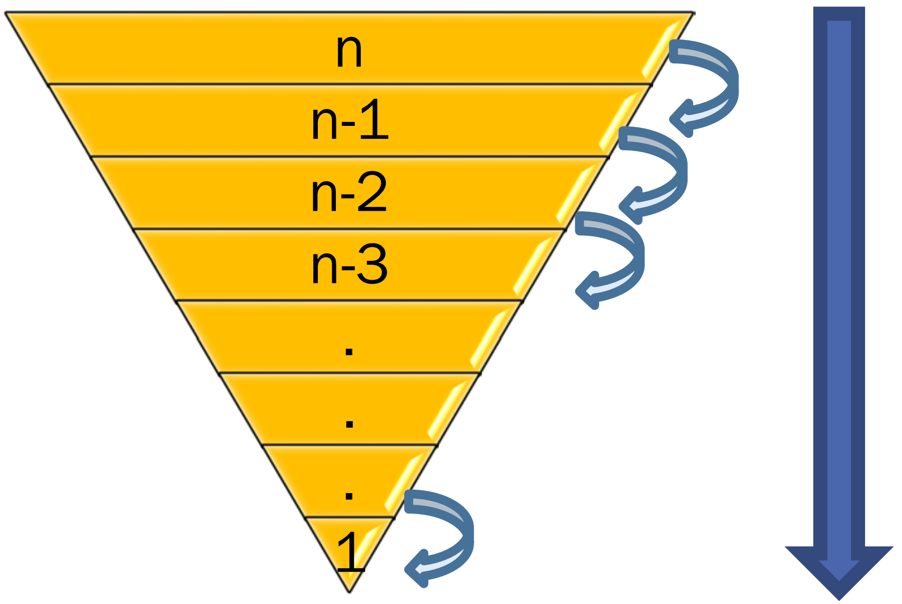
Decrease-and-Conquer

- Reduce problem instance to smaller instance of the same problem and solve smaller instance
- Extend solution of smaller instance to obtain solution to original instance

- Can be implemented:
  - *Top-down (recursive)*
  - *Bottom-up (iterative)*

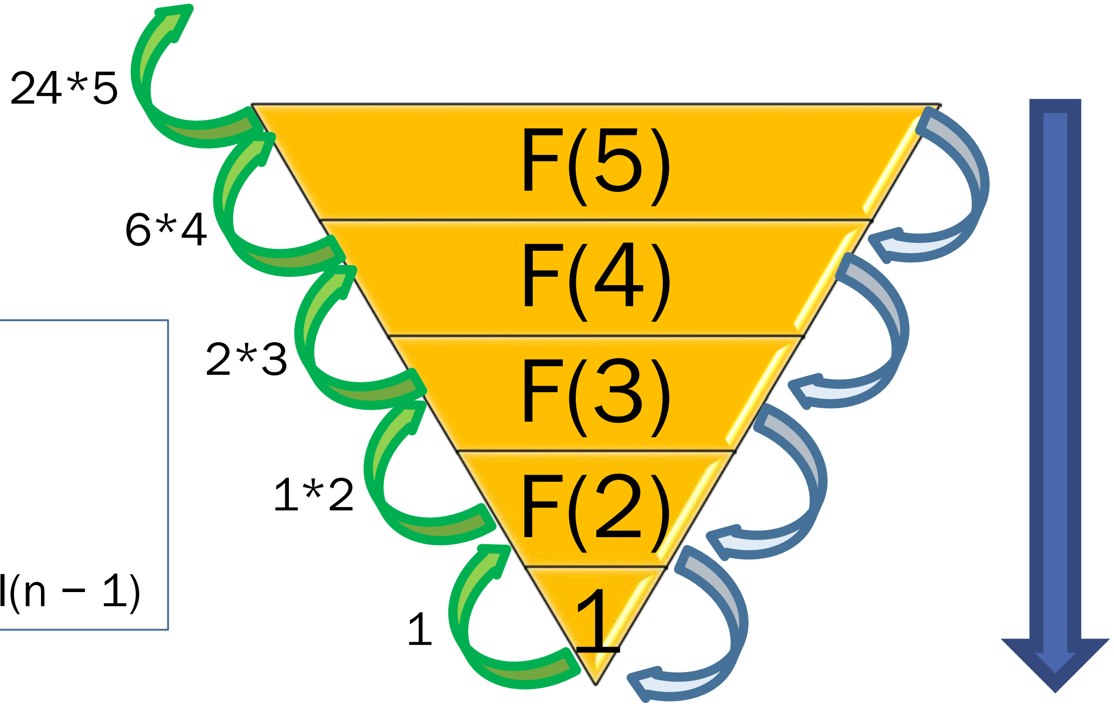
Decrease-and-Conquer

- Top-down (recursive):



Example: top-down (recursive)

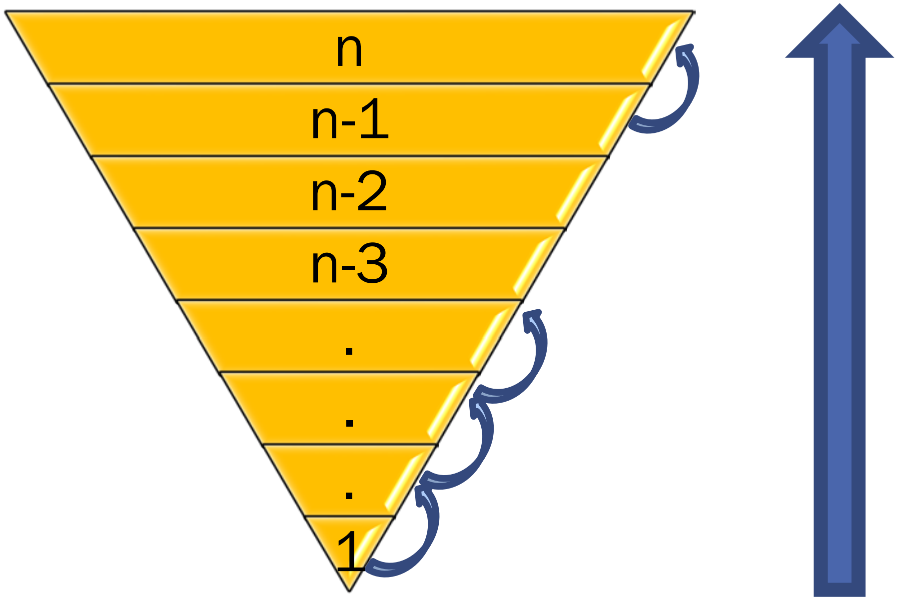
```
Factorial(n)
    if n=0 or n=1 then
        return 1
    else
        return n * Factorial(n − 1)
```



Factorial (5)= ?

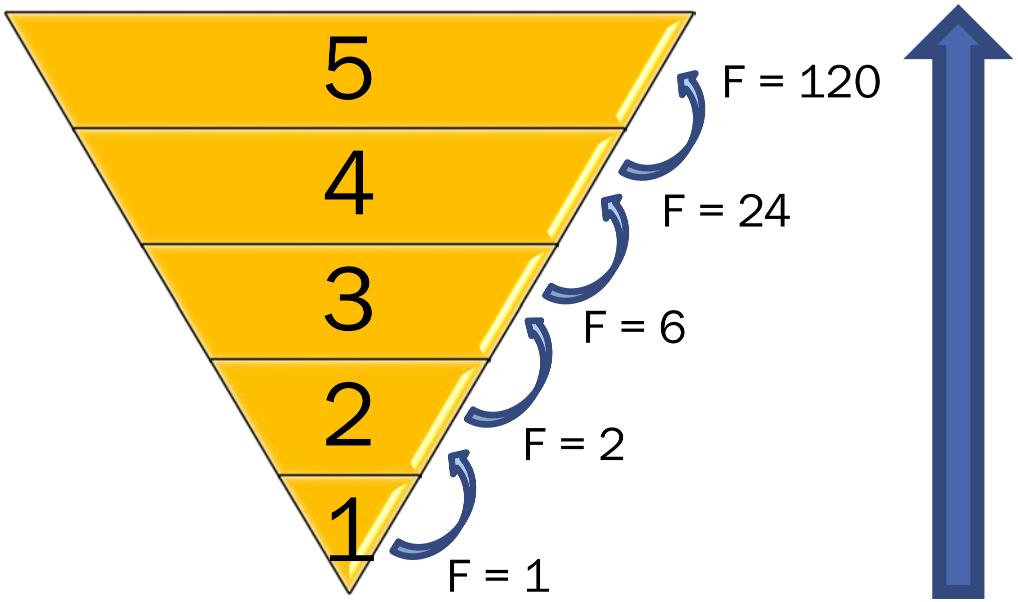
Decrease-and-Conquer

- Bottom-up (iterative):



Example: bottom-up (iterative)

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

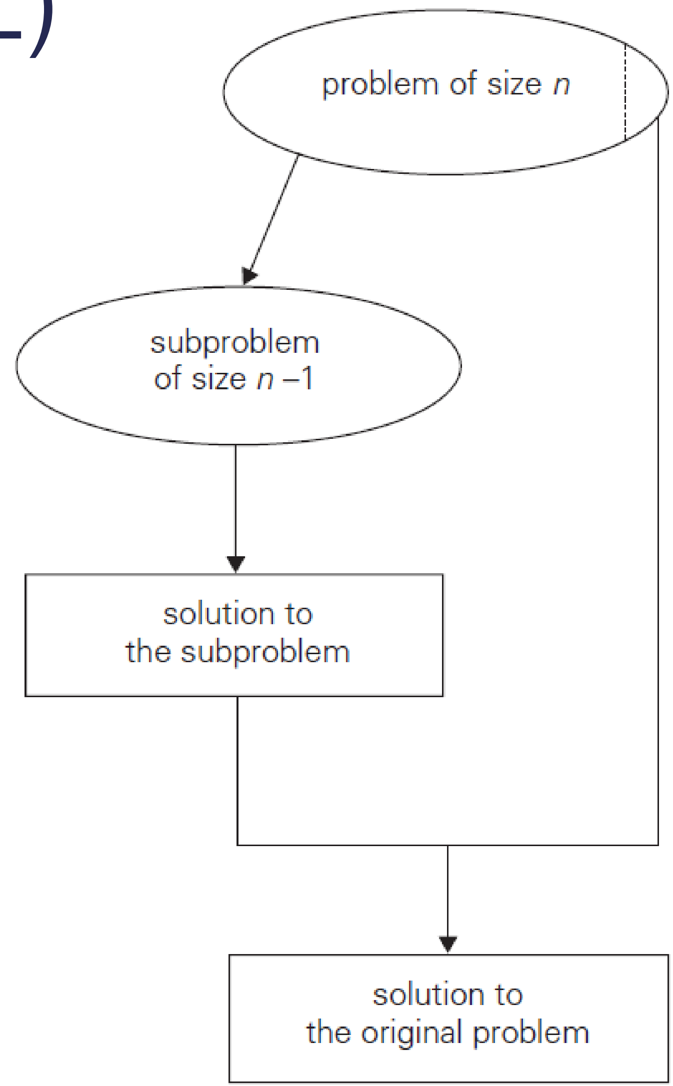


Factorial (5)= ?

Three types of Decrease and Conquer

- Decrease by a constant (usually by 1)
  - *1.1 Insertion sort*
  - *1.2 Generating permutations*
  - *1.3 Generating subsets*
- Decrease by a constant factor (usually by half)
  - *2.1 Binary search*
  - *2.2 Exponentiation by squaring*
  - *2.3 Fake coin problem*
- Variable-size decrease
  - *3.1 Euclid's algorithm*

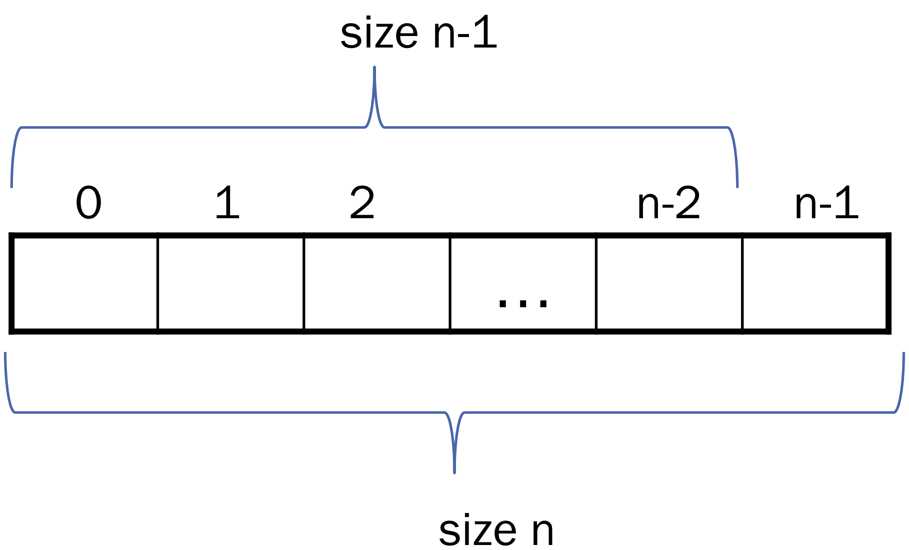
Decrease by a constant (usually by 1)

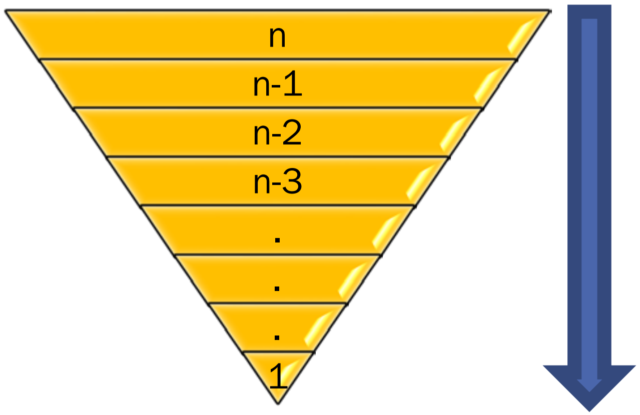


1.1 INSERTION SORT

1.1 Insertion sort

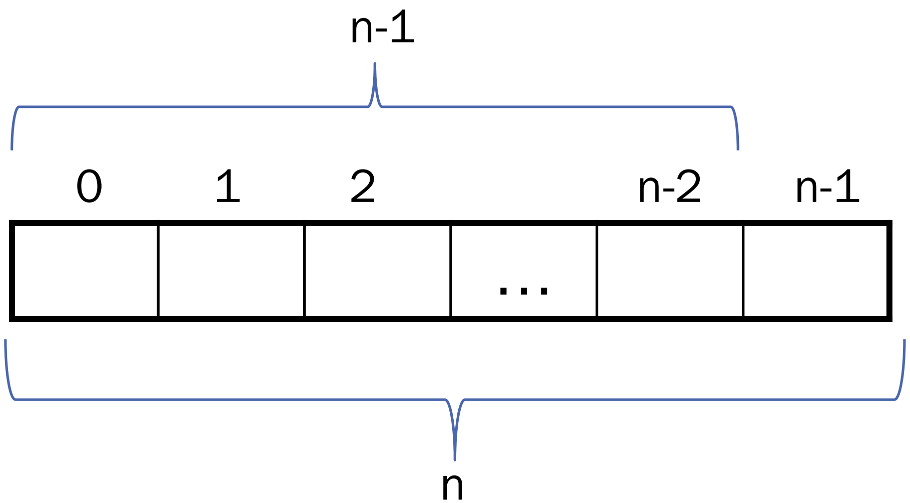
- Insertion sort (A[0..n-1])
  1. *Sort A[0..n-2]*
  2. *Insert A[n-1] in its proper place among the sorted A[0..n-2]*





Top-down (recursive)

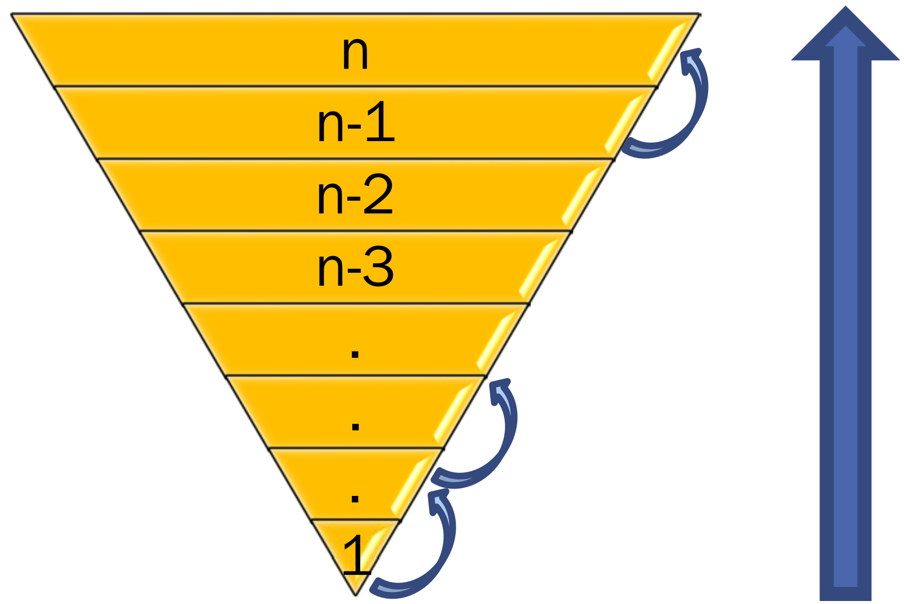
1.1 Insertion sort (recursive)




```
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
```

1.1 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

1.2 GENERATING PERMUTATIONS

1.2 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*

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:*

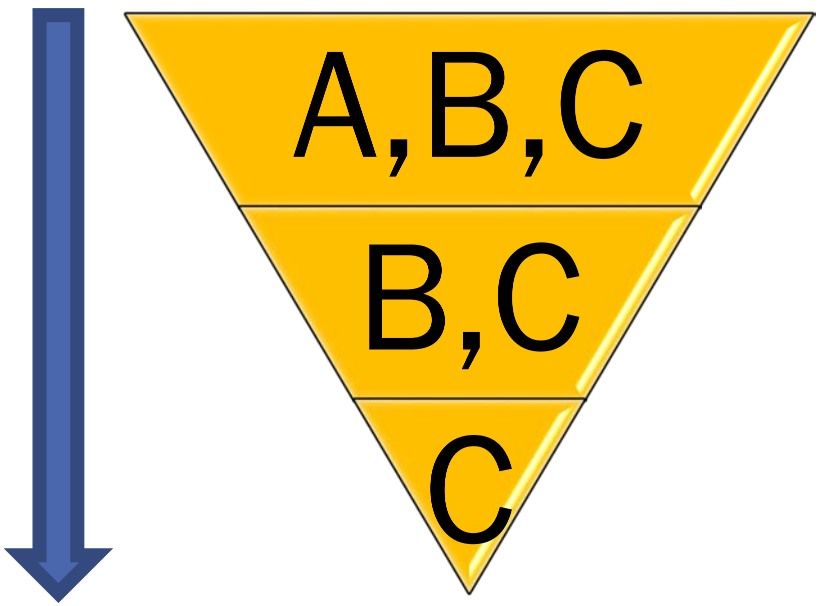
    B C and C B

  - *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



ABC BAC BCA ACB CAB CBA

BC   CB

C

1.2 Generating permutations

```
generatePermutation (a_1, a_2, ..., a_n)
If n>1
     Permutations = generatePermutation (a_1, a_2, ..., a_n-1)
     for each p in Permutations
           insert a_n before a_1 and add to newPermutations
           for i ← 1 to n-1
              insert a_n after a_i and add to newPermutations
      return newPermutations
```

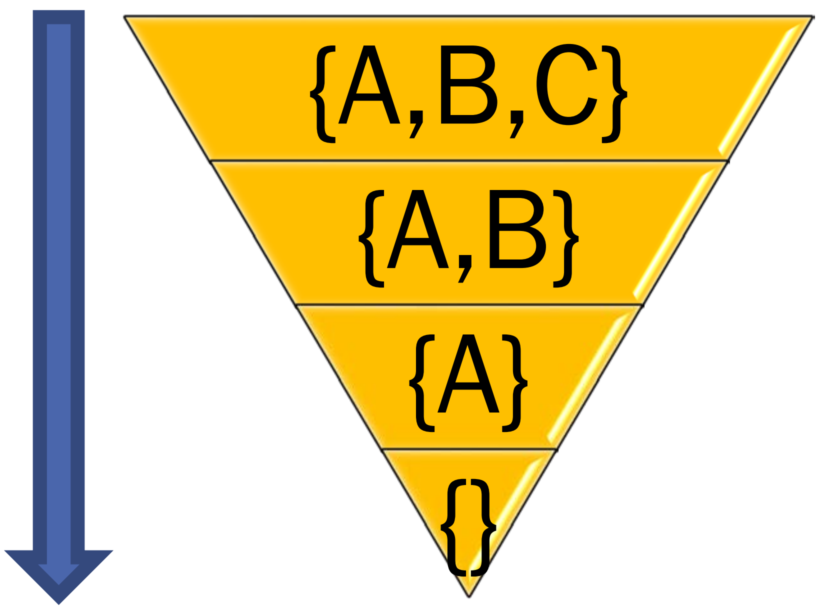
1.3 GENERATING SUBSETS

1.3 Generating subsets

- To find all subsets of n objects:
  - *Find all subsets of n-1 of those objects*
  - *For each subset, copy it and insert the remaining object into the copy subset*

1.3 Generating subsets

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



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

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

{} {A}

{}

1.3 Generating subsets

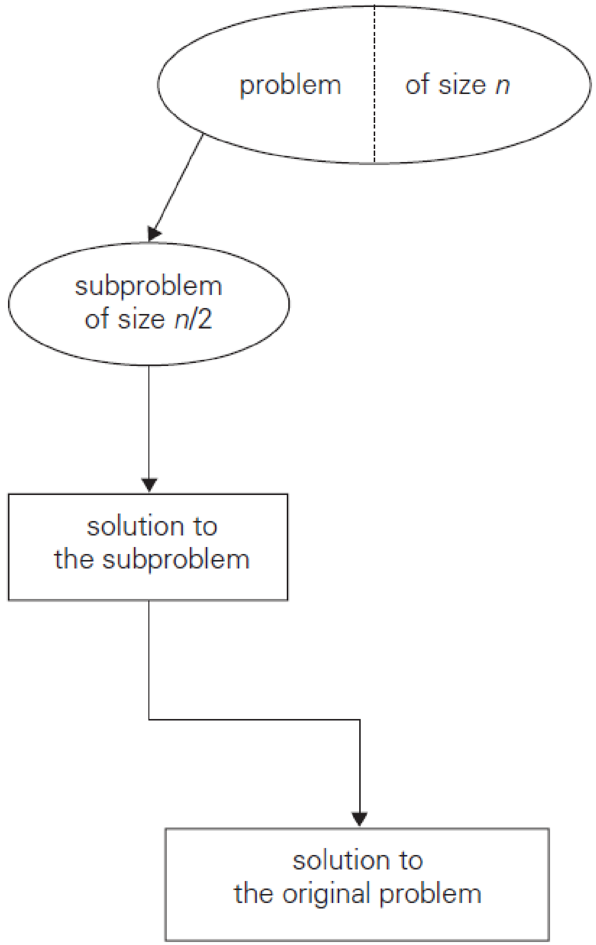
```
generateSubsets (a1, a2, ..., an)
If n>0
     subsets = generateSubsets (a1, a2, ..., an-1)
     for each subset s in subsets
           clone s to s'
           insert an to s'
```

DECREASE BY A CONSTANT FACTOR

Decrease by a constant factor (usually by half)

- Make the problem smaller by some constant factor

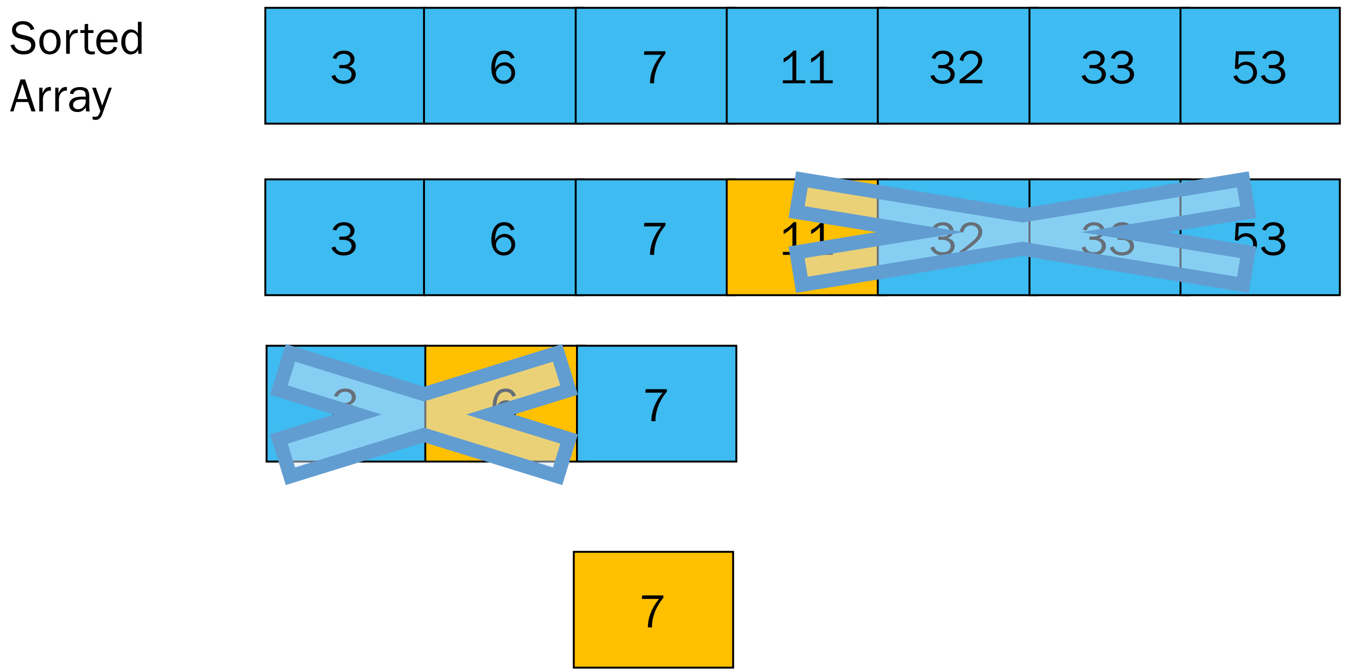
- Typically the constant factor is *two*, i.e, we divide the problem in half



2.1 BINARY SEARCH

2.1 Binary search

- Example: binary search, key =7



2.1 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

2.1 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
```

2.1 Binary search

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

- Example: Binary search, 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

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

2.1 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 $\lceil \log_2 n \rceil + 1$)

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
```

EXPONENTIATION BY SQUARING

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*

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

2.2 Exponentiation by squaring

- Compute a^n where n is a nonnegative integer

```
power(a, n):
    if (n =1)
        return a
    if (n % 2 = 0)
        return power(a * a, n / 2)
    else:
        return a * power(a * a, (n - 1) / 2)
```

Efficiency = O(log n)

FAKE COIN PROBLEM

2.3 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.

2.3 Fake coin problem (solution)

- Divide the coins into two equal piles. If n is odd, set one coin aside first. Compare the piles (*i.e.*, 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 lighter pile.
- Discard the heavier pile. Using the lighter pile, repeat the above procedure until there are only two coins, or the fake coin has been found.
- If there are only two coins left, the lighter of the two is the fake.

2.3 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 base: 1 weight comparison is needed.
- Worst case: 4 weight comparisons are needed. $\lfloor \log_2 n \rfloor$

2.3 Fake coin 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
```

2.3 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)

VARIABLE SIZE DECREASE

3.1 Euclid's algorithm

- Problem: Find gcd(m,n), the greatest common divisor of two nonnegative numbers

- Examples: gcd(8,6) = 2
  - *Divisors of 6 are 1, 2, 3, 6*
  - *Divisors of 8 are 1, 2, 4, 8*

3.1 Euclid's algorithm

- Example gcd(60,24) = ?
- Euclid's algorithm:
- $$\gcd(m,n) = \gcd(n, m \bmod n)$$
- until the second number becomes 0
- gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

3.1 Euclid's algorithm (recursive) pseudocode

```
GCD (m, n)
    if ((m % n) = 0)
        return n
    else
        return GCD(n, m % n)
```

3.1 Euclid’s algorithm (iterative) pseudocode

```
GCD(m,n)
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

BONUS PROBLEM (BRUTE FORCE)

Kaprekar’s constant

- Take any four-digit number that has at least two different digits. (Can't have 1111 or 2222 etc.) The number can have leading zeros.
- Rearrange the digits to make the largest possible number you can make with them, and also the smallest possible number.
- Subtract these two numbers.
- Repeat steps 2 and 3.

- This process *always leads to 6174.*

Example: 3141

- 4311 - 1134 = 3177
- 7731 - 1377 = 6354
- 6543 - 3456 = 3087
- 8730 - 0378 = 8352
- 8532 - 2358 = 6174
- 7641 - 1467 = 6174
- …

The problem(s)

- The Internet says the preceding process will reach 6174 in at most 7 steps for all the numbers up to 9998.
- Problem 1:
  - *Write a brute-force algorithm to verify this, and (for bonus adventures) implement some actual code.*
  - *Watch out for infinite loops!*
- Problem 2:
  - *Have your program output ALL of the starting numbers that require 7 steps (or, if the Internet lied, all the numbers that require the maximum # of steps).*

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*