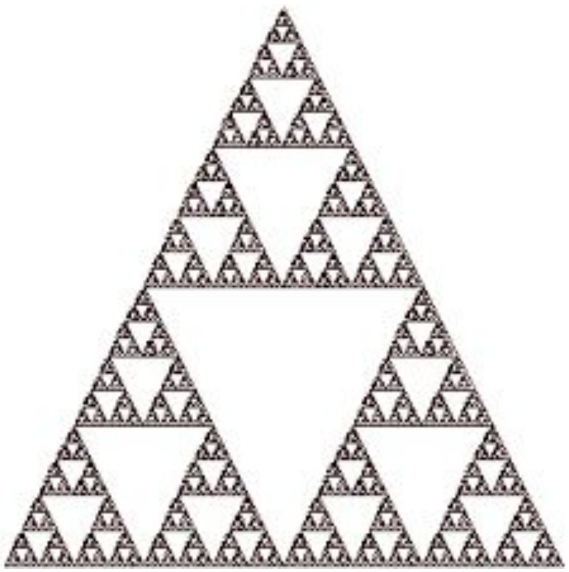
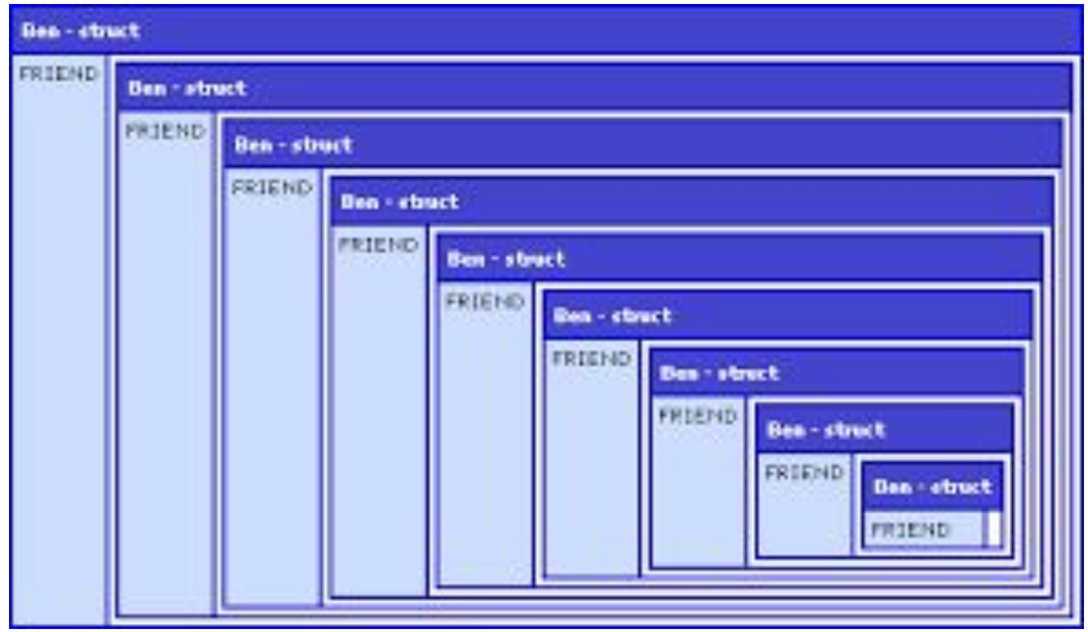
Recursion

Recursion: Basic idea

- We have a bigger problem whose solution is difficult to find
- We divide/decompose the problem into smaller (sub) problems
  - Keep on decomposing until we reach to the smallest sub-problem (base case) for which a solution is known or easy to find
  - Then go back in reverse order and build upon the solutions of the sub-problems
- Recursion is applied when the solution of a problem depends on the solutions to smaller instances of the same problem




Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND
Ben - struct
FRIEND

Example 1: Factorial

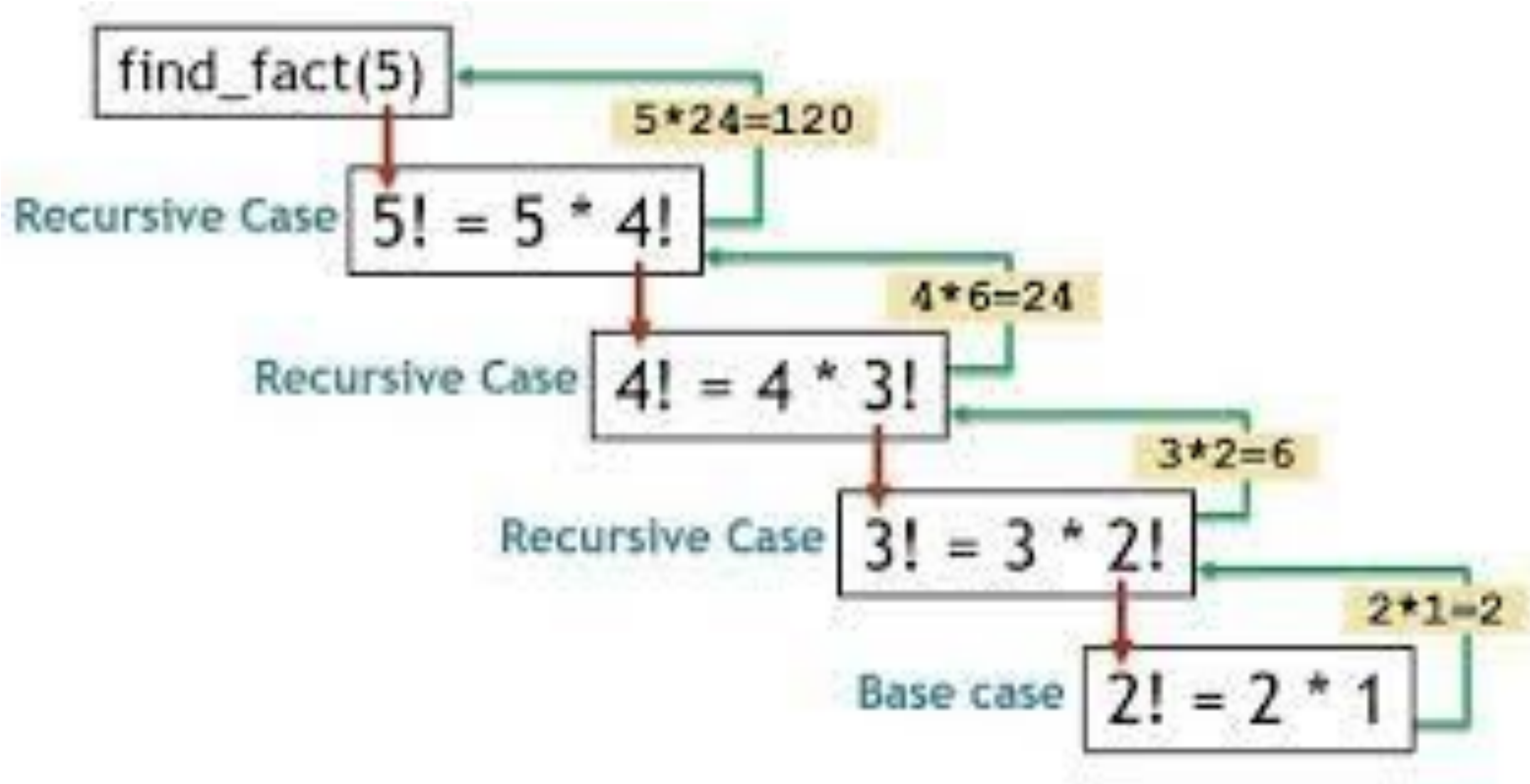
- A function which calls itself

```
int factorial ( int n ) {
    if ( n == 0)    // base case
        return 1;
    else            // general/ recursive case
        return n * factorial ( n - 1 );
}
```

Finding a recursive solution

- Each successive recursive call should bring you closer to a situation in which the answer is known (cf. n-1 in the previous slide)
- A case for which the answer is known (and can be expressed without recursion) is called a base case
- Each recursive algorithm must have at least one base case, as well as the general recursive case

- The factorial of a positive integer n, **denoted $n!$,** is defined as the product of the integers from 1 to n. For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.



Recursion in Action: *factorial(n)*

```
factorial (5) = 5 x factorial (4)
              = 5 x (4 x factorial (3))
              = 5 x (4 x (3 x factorial (2)))
              = 5 x (4 x (3 x (2 x factorial (1))))
              = 5 x (4 x (3 x (2 x (1 x factorial (0)))))
              = 5 x (4 x (3 x (2 x (1 x 1))))
              = 5 x (4 x (3 x (2 x 1)))
              = 5 x (4 x (3 x 2))
              = 5 x (4 x 6)
              = 5 x 24
              = 120
```

Base case arrived
Some concept from elementary maths: Solve the inner-most bracket, first, and then go outward

Recursion vs. Iteration: Computing N!

- The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.
  - Iterative Solution

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

  - Recursive Solution

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

Recursion: Do we really need it?

- In some programming languages recursion is imperative
  - For example, in declarative/logic languages (LISP, Prolog etc.)
  - Variables can’t be updated more than once, so no looping
  - Heavy backtracking

Recursion in Action: *factorial(n)*

```
factorial (5) = 5 x factorial (4)
              = 5 x (4 x factorial (3))
              = 5 x (4 x (3 x factorial (2)))
              = 5 x (4 x (3 x (2 x factorial (1))))
              = 5 x (4 x (3 x (2 x (1 x factorial (0)))))
              = 5 x (4 x (3 x (2 x (1 x 1))))
              = 5 x (4 x (3 x (2 x 1)))
              = 5 x (4 x (3 x 2))
              = 5 x (4 x 6)
              = 5 x 24
              = 120
```

Base case arrived

Some concept from elementary maths: Solve the inner-most bracket, first, and then go outward

How to write a recursive function?

- Determine the size factor (e.g. *n* in *factorial(n)*)
- Determine the base case(s)
  - the one for which you know the answer (e.g. 0! = 1)
- Determine the general case(s)
  - the one where the problem is expressed as a smaller version of itself (must converge to base case)
- Verify the algorithm
  - use the "Three-Question-Method" – next slide

Linear Recursion

- The simplest form of recursion is ***linear recursion***, where a method is defined so that it makes at most one recursive call each time it is invoked
- This type of recursion is useful when we view an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set

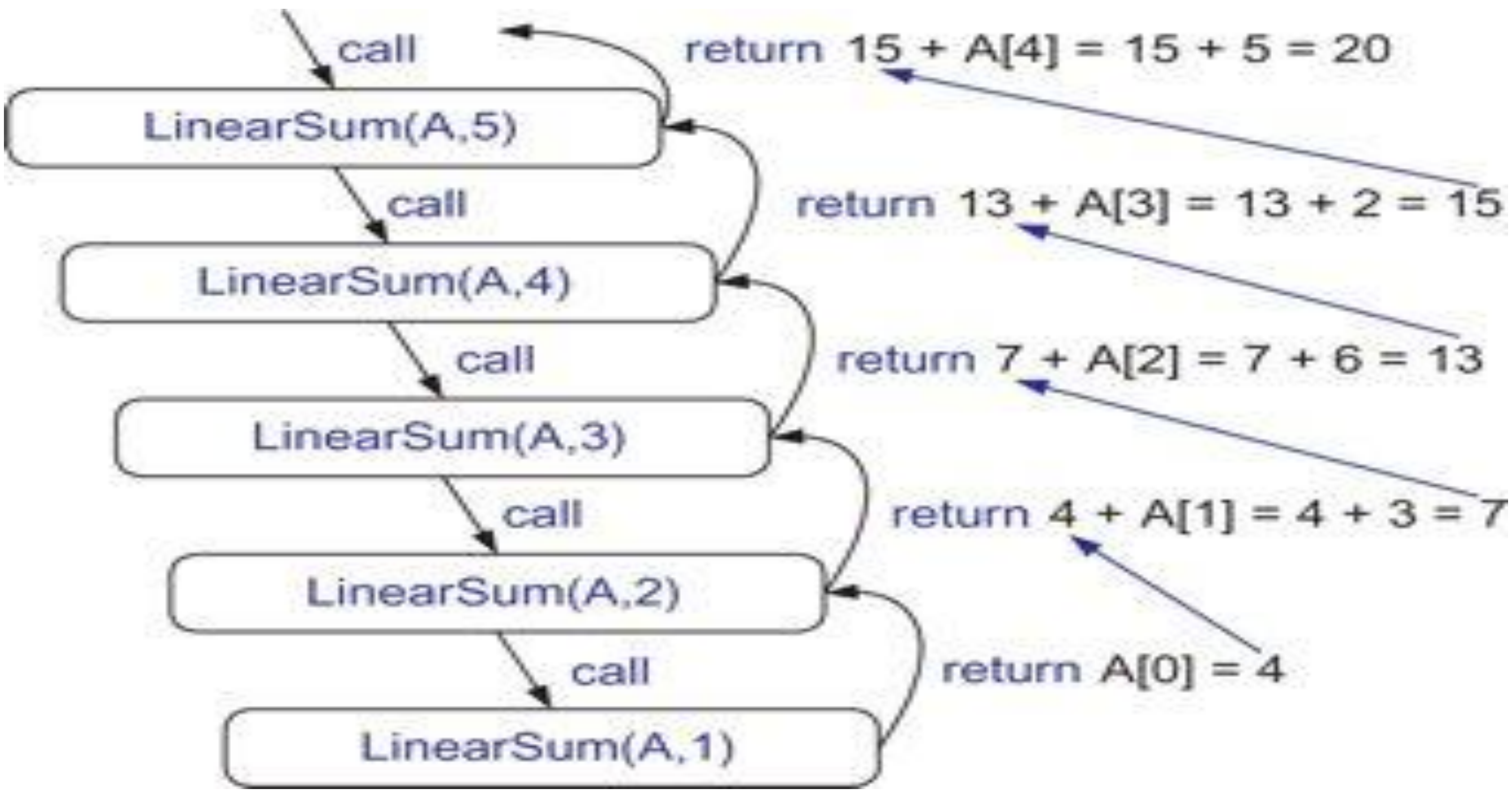
Example 2: Summing the Elements of an Array

- We can solve this summation problem using linear recursion by observing that the sum of all n integers in an array A is:
  - Equal to $A[0]$, if $n = 1$(The array has one element), or
  - The sum of the first $n - 1$ integers in A plus the last element

```
int LinearSum(int A[], n){
    if n = 1 then
        return A[0];  // base case
    else
        return A[n-1] + LinearSum(A, n-1)
}
```

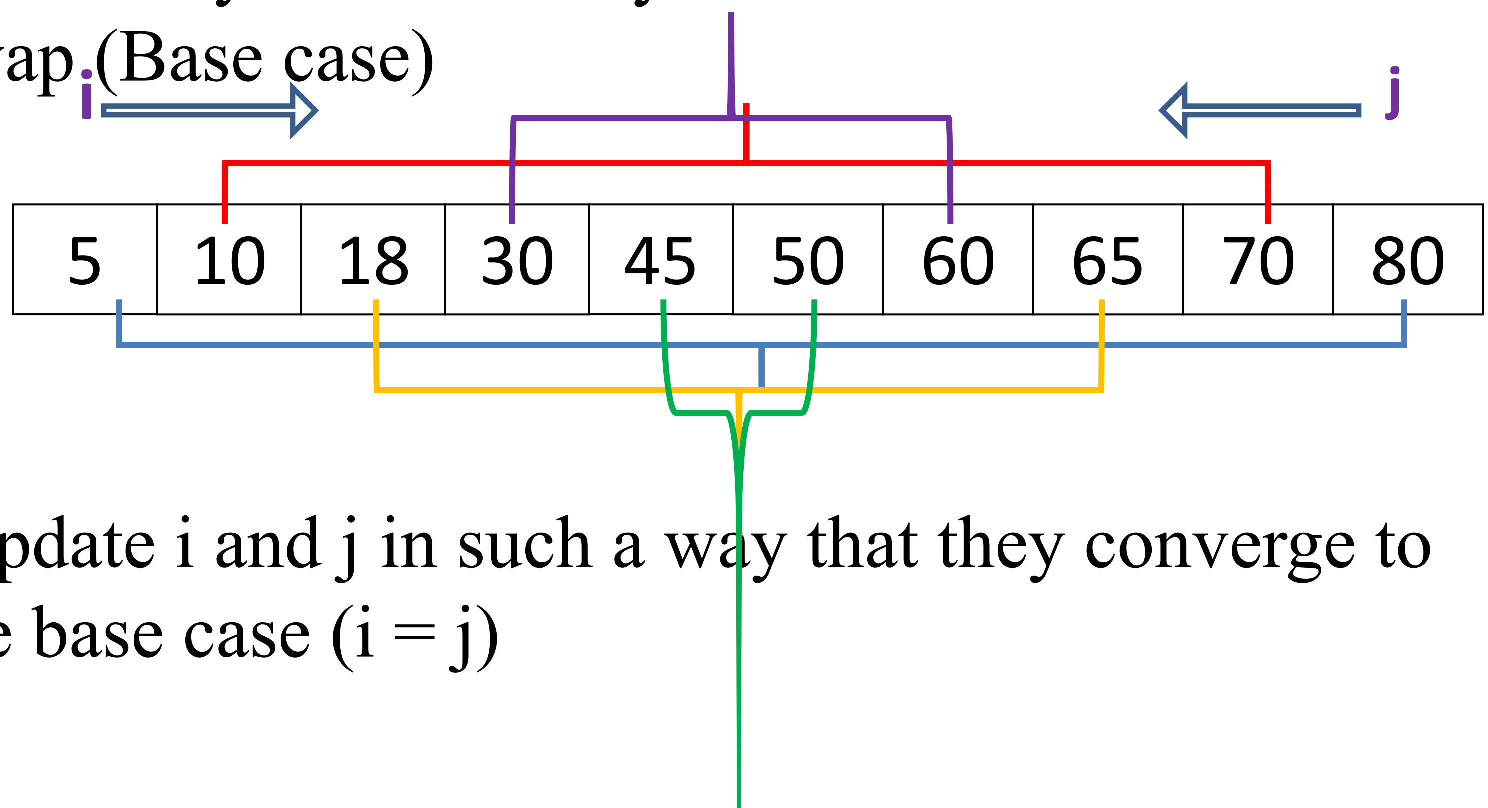
Analyzing Recursive Algorithms using Recursion Traces

- Recursion trace for an execution of *LinearSum(A,n)* with input parameters *A* = [4,3,6,2,5] and *n* = 5



Linear recursion: Reversing an Array

- Swap 1st and last elements, 2nd and second to last, 3rd and third to last, and so on
- If an array contains only one element no need to swap.(Base case)



- Update i and j in such a way that they converge to the base case (i = j)

Example 3: Reversing an Array

```
void reverseArray(int A[], i, j){
   if (i < j){
      int temp = A[i];
      A[i] = A[j];
      A[j] = temp;
      reverseArray(A, i+1, j-1)
   }
   // in base case, do nothing
}
```

Linear recursion: run-time analysis

- Time complexity of linear recursion is proportional to the problem size
  - Normally, it is equal to the number of times the function calls itself
- In terms of Big-O notation time complexity of a linear recursive function/algorithm is *O(n)*