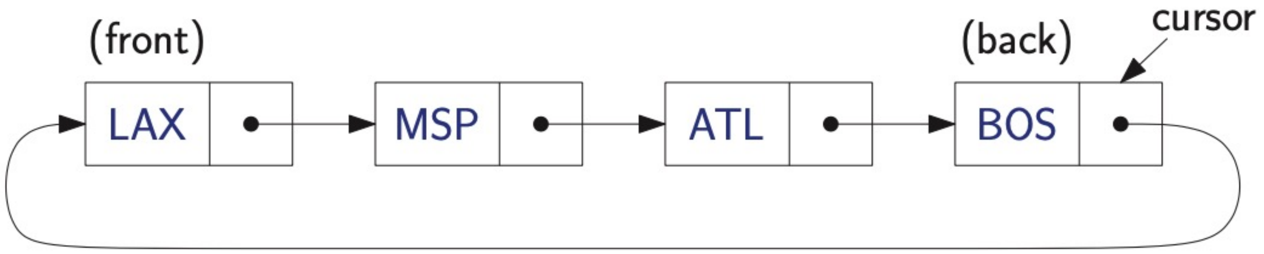
CIRCULARLY LINKED LIST: IMPLEMENTATION

- The method “add” puts *e* in a node and adds it to the beginning of the list
- i.e., if "N" denotes the node that "cursor" points to, then the method “add” creates a new node whose element equals "e", and inserts it right after N.

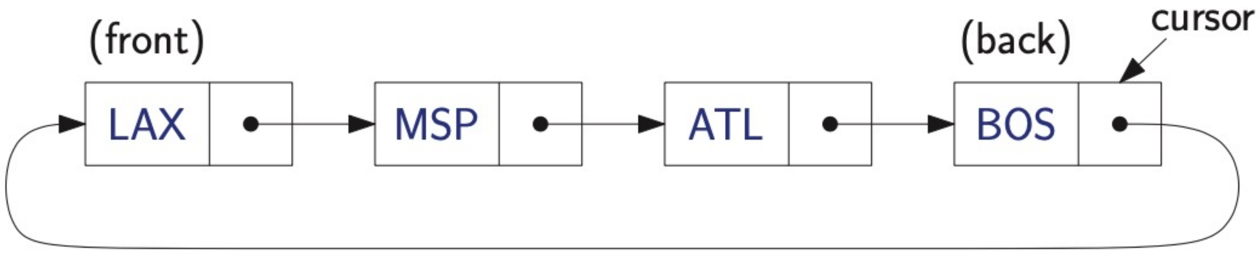
```
void CircleList::add(const Elem& e) {
    CNode* v = new CNode; // create a new node
    v->elem = e;
    if (cursor == NULL){ // handling the special case when the list is empty
        v->next = v; // v points to itself
        cursor = v;
    }
    else { // if the list is not empty
        v->next = cursor->next;
        cursor->next = v;
    }
}
```



CIRCULARLY LINKED LIST: IMPLEMENTATION

- The method "remove" deletes the node at the beginning of the list.
- i.e., if "N" denotes the node that "cursor" points to, then the method "remove" deletes the node right after N (unless N was the only node in the list, in which case N itself is deleted).

```
void CircleList::remove() {
    CNode* old = cursor->next; // the node being removed
    if (old == cursor) { // if "true", it implies that the list has only one node
        cursor = NULL; // indicating that the list is now empty
    }
    else { // if the list does not contain just a single node
        cursor->next = old->next;
    }
    delete old;
}
```



CIRCULARLY LINKED LIST: PLAYLIST EXAMPLE

To help illustrate the use of our CircleList implementation, consider how building a simple interface of a playlist in a music player. Here, the track at the curser is marked with a star (*)

```
int main() {
    CircleList playList;                // [ ]  (the list is empty)
    playList.add("Faint");              // [Faint*]
    playList.add("Numb");               // [Numb, Faint*]
    playList.add("In the End");         // [In the End, Numb, Faint*]
    playList.advance();                 // [Numb, Faint, In the End*]
    playList.advance();                 // [Faint, In the End, Numb*]
    playList.remove();                  // [In the End, Numb*]
    playList.add("Castle Of Glass");    // [Castle Of Glass, In the End, Numb*]
    return EXIT SUCCESS;
}
```

45

RECURSION

RECURSION

- Another way to achieve repetition, other than loops, is through recursion
- Recursion is when a function repeatedly calls itself
    - until it reaches a **base case**

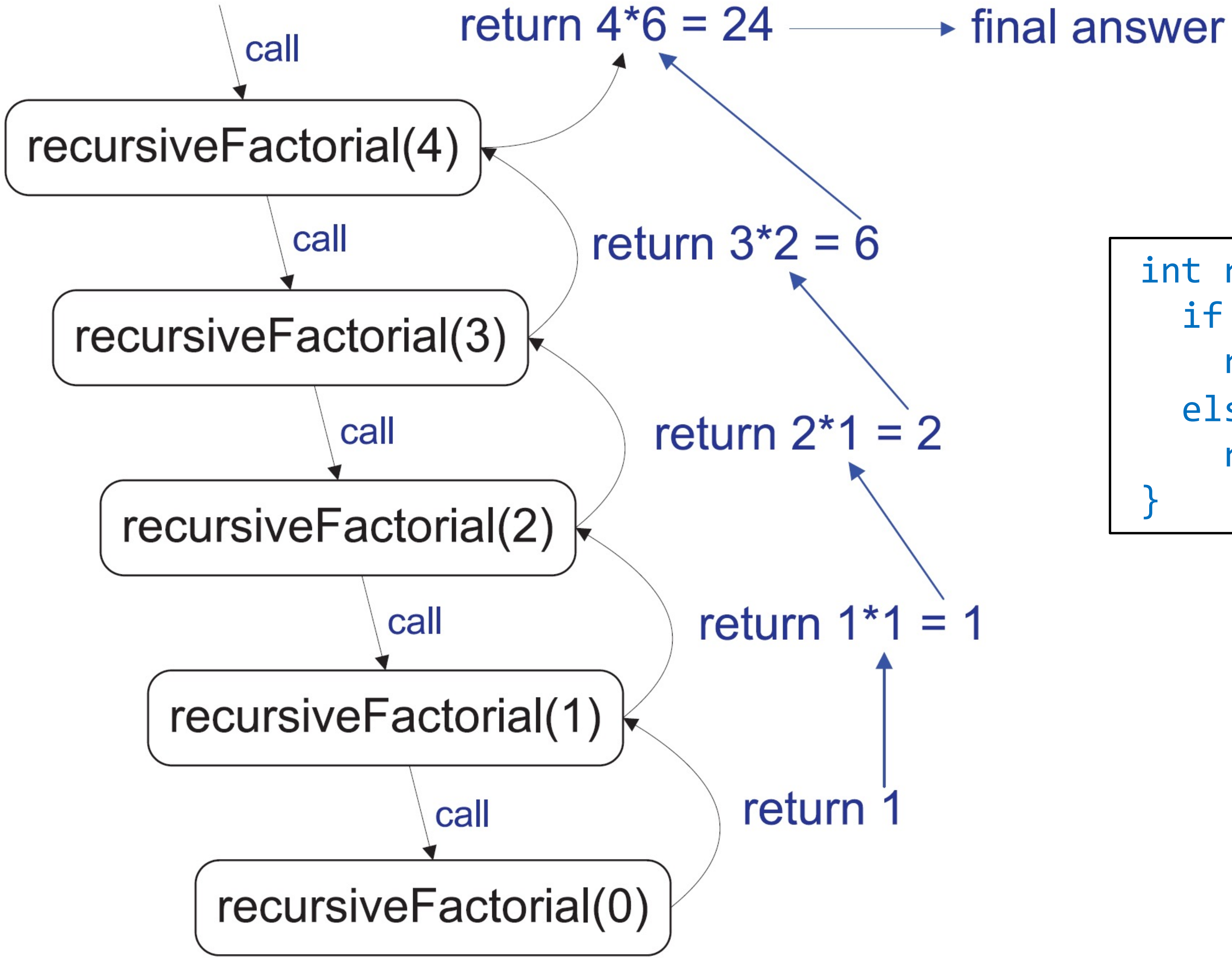
EXAMPLE: FACTORIAL USING RECURSION

- The factorial, n!, is defined as:
  - If n > 0 then n! = 1 x 2 x 3 x 4 x 5 x .... x n ≡ (n-1)! * n
  - If n = 0 then 0! = 1

How would you implement this recursively in C++?

```
int recursiveFactorial(int n) {
   if (n == 0) // checks if we reach the base case
       return 1;
   else // if not, make a recursive call
       return n * recursiveFactorial(n-1);
}
```

EXAMPLE: FACTORIAL – RECURSIVE TRACE




```
int recursiveFactorial(int n){
  if (n == 0)
    return 1;
  else
    return n*recursiveFactorial(n-1);
}
```

EXAMPLE: LINEAR SUM USING RECURSION

Algorithm LinearSum(A, n):

Input: Array A and integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

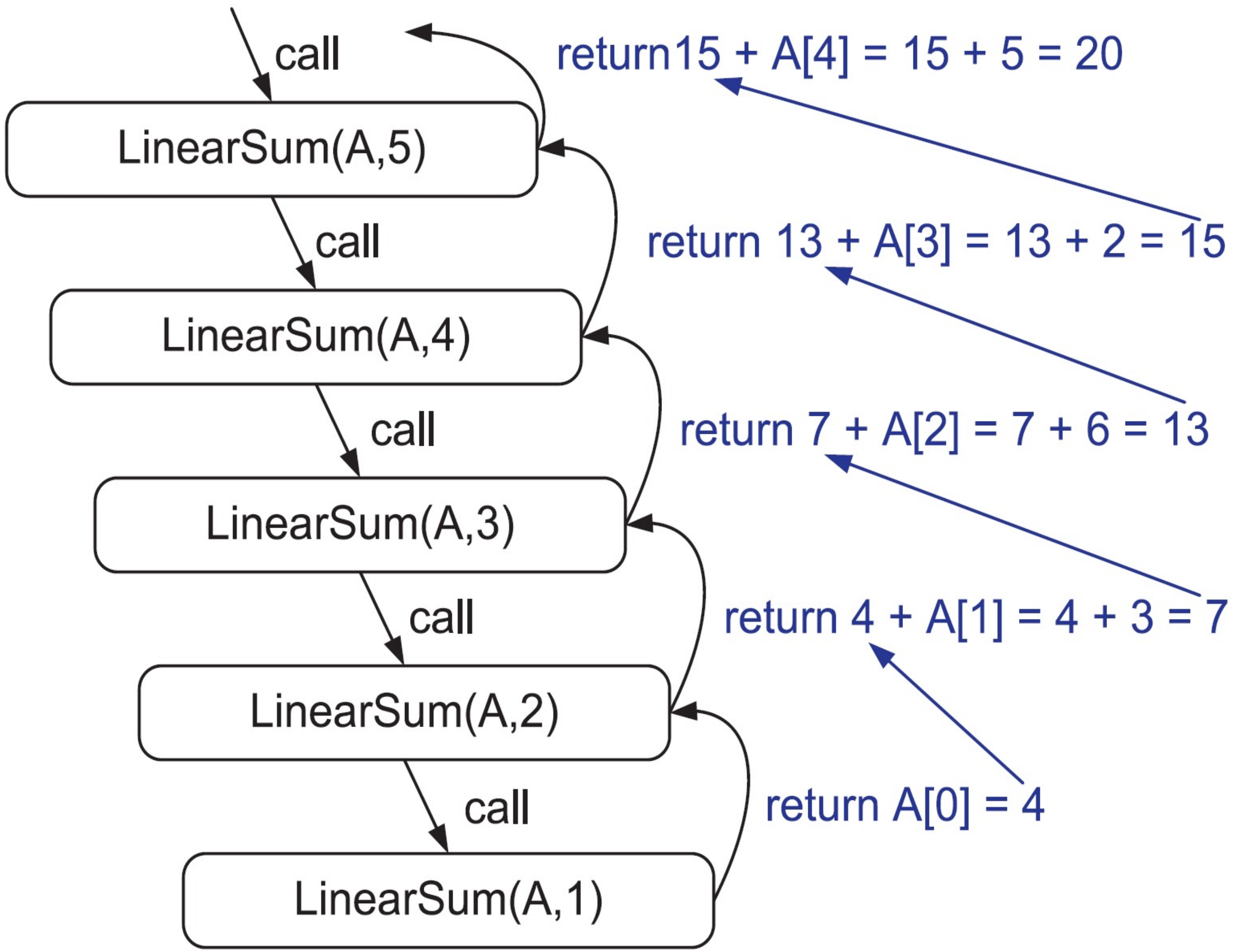
How would you implement this recursively in C++?

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

e.g. A =

0	1	2	3	4
4	3	6	2	5

EXAMPLE: LINEAR SUM – RECURSIVE TRACE




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

In this example:
A = {4,3,6,2,5} and *n*=5

EXAMPLE: REVERSING AN ARRAY

Algorithm ReverseArray(*A, i, j*):

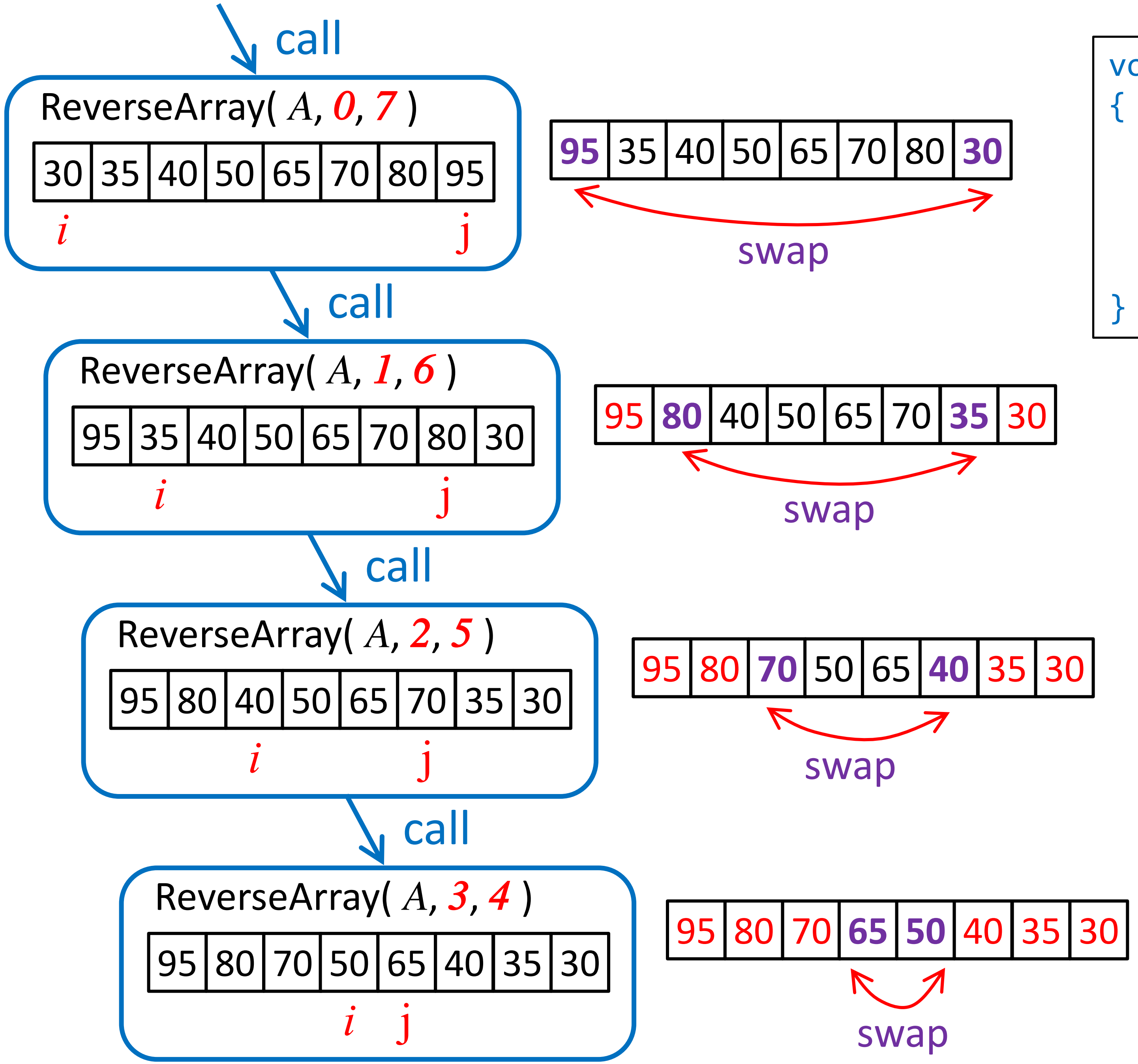
Input: Array *A* and non-negative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

How would you implement this recursively?

i							*j*
30	35	40	50	65	70	80	95

```
void ReverseArray(int *A, int i, int j){
  if(i < j){
        swap(A, i, j); //you define a swap function before
        ReverseArray(A, i+1, j-1);
    }
}
```




```
void ReverseArray(int *A, int i, int j)
{
  if(i < j){
        swap(A, i, j);
        ReverseArray(A, i+1, j-1);
    }
}
```

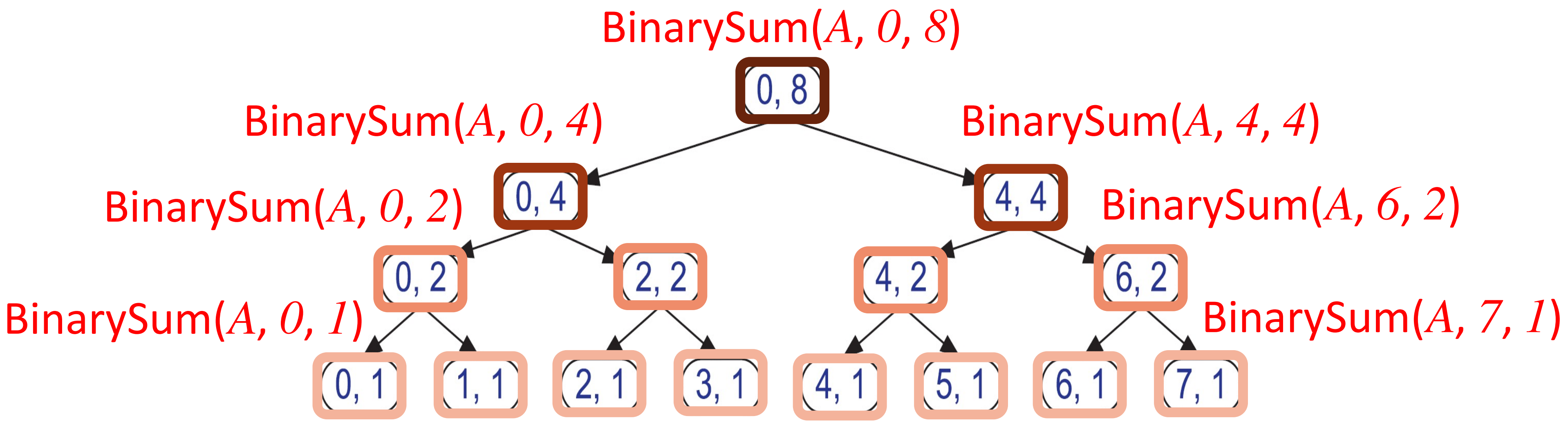
```
void swap(int *A, int i, int j){
   int temp;
   temp = A[i];
   A[i] = A[j];
   A[j] = temp;
}
```

BINARY RECURSION

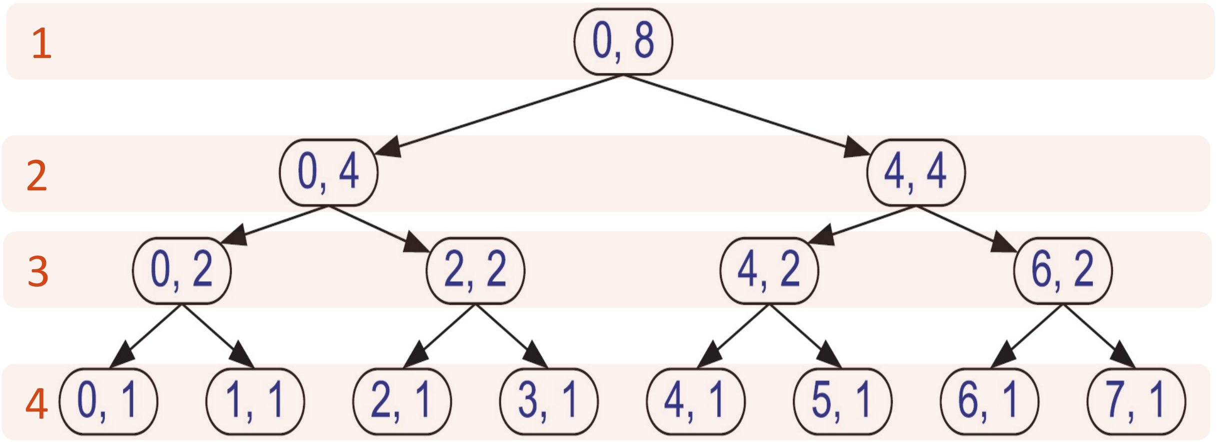
- When an algorithm makes two recursive calls, we say it uses **binary recursion**
- For example, these two calls can be used to solve **two similar halves of a problem**
- Let us revisit the problem of *summing the n elements of an integer array*...

EXAMPLE: BINARY RECURSION – SUM

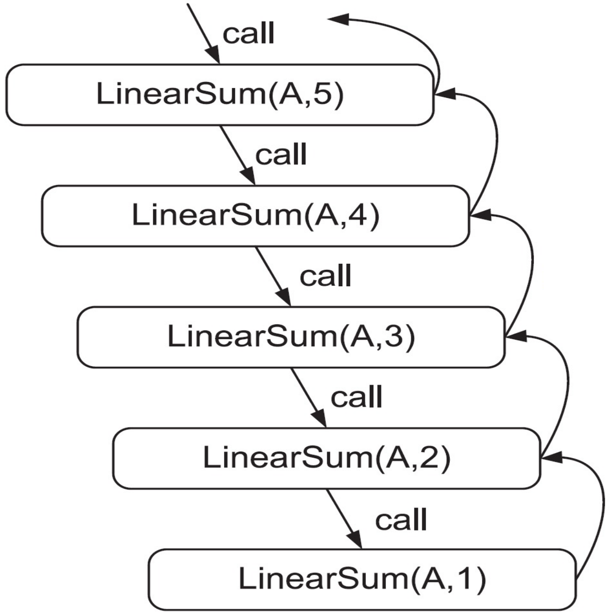
Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
        if $n = 1$ **then**
            return $A[i]$
        return BinarySum(A, i, $\lceil n/2 \rceil$) + BinarySum(A, $i+\lceil n/2 \rceil$, $\lfloor n/2 \rfloor$)



EXAMPLE: BINARY RECURSION – SUM



In this example, the maximum number of iterations we need to sum up 8 elements is $1+\log_2 n$, which is at most 4.



This is a big improvement over the number of iterations needed by “LinearSum”, where you need *n* iterations for *n* elements.