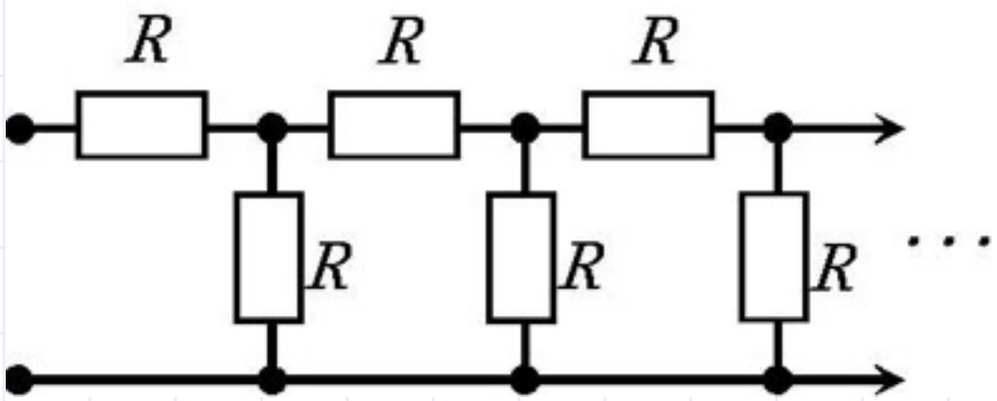
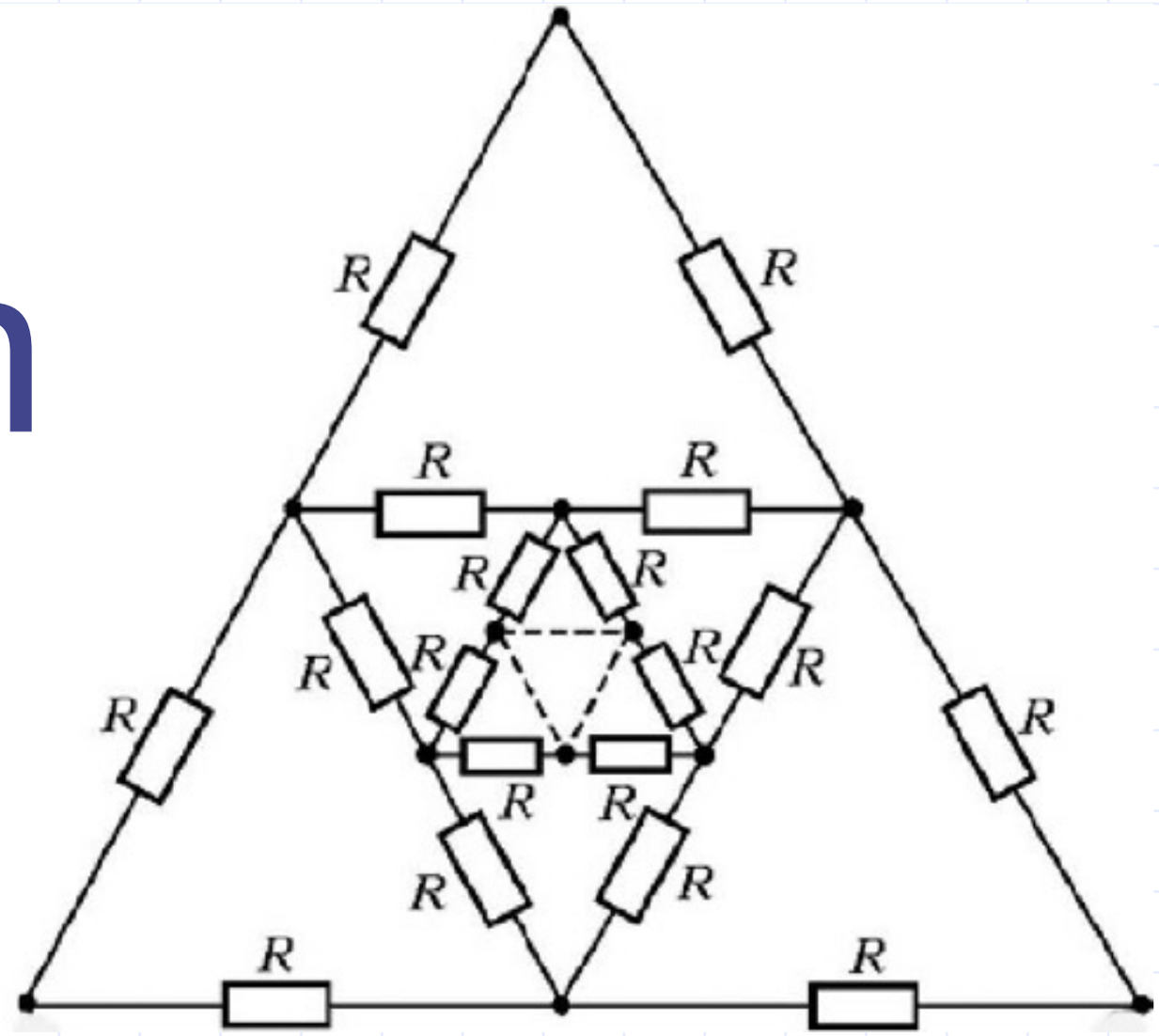
ESC101: Introduction to Computing

Recursion





Searching in an Array

- We can have other recursive formulations
- Search1: find_key (a, start, end, key)
  - Search key between a[start]…a[end]

```
if (start > end) return 0;
if (a[start] == key) return 1;
return find_key(a, start+1, end, key);
```

Example 2: In-place reversing an array

Write a function reverse(int a[], int start, int end) that reverses the values contained in the first n indices of a[]. That is,
a[start] and a[end] are exchanged, a[start+1] and a[end-1] are exchanged, and so on.

reverse (a,start, end): formulating the problem recursively

Basic idea:

1. if end==start or start>end, return. Nothing to reverse.
2. Otherwise,
   a) exchange a[start] with a[end].
   b) call reverse on array starting at position start+1 and with end being end-1.

Let's write this…

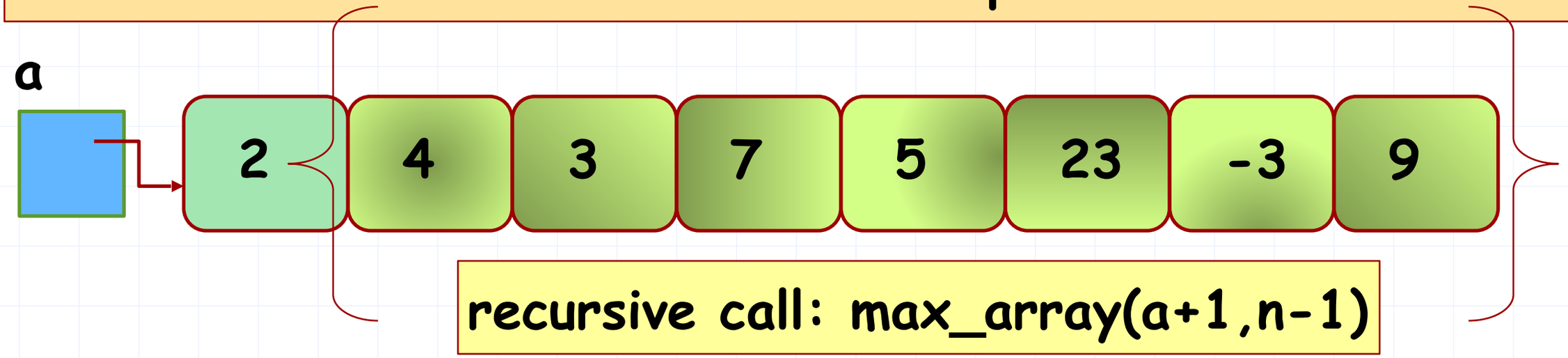
```
void reverse(int a[], int start, int end) {
   if (start==end || start>end ) return ;
   else {
         swap(a,start,end);
         reverse(a,start+1, end-1);
   }
}

void swap( int a[], int n1, int n2)
{
   int tmp=a[n1];
   a[n1] = a[n2];
   a[n2] = tmp;
}

int main()
{
   int arr[]={100,10,4,20,45,56,72,43,33,93};
   for( int i=0; i<10; i++)
      printf("%d ",arr[i]);
   printf("\n");
   reverse(arr, 0,9);
   for( int i=0; i<10; i++)
      printf("%d ",arr[i]);
   printf("\n");
   return 0;
}
```

```
void reverse(int a[], int start, int end) {
   if (start==end || end-start==1 ) return ;
   else {
         swap(a,start,end-1);
         reverse(a,start+1, end-1);
   }
}

void swap( int a[], int n1, int n2)
{
   int tmp=a[n1];
   a[n1] = a[n2];
   a[n2] = tmp;
}

int main()
{
   int arr[]={100,10,4,20,45,56,72,43,33,93};
   for( int i=0; i<10; i++)
      printf("%d ",arr[i]);
   printf("\n");
   reverse(arr, 0,10);
   for( int i=0; i<10; i++)
      printf("%d ",arr[i]);
   printf("\n");
   return 0;
}
```

Example 3: Array Maximum

Find the maximum of the numbers in an array.

```
int max_array(int a[], int n);
If n is of size 0 then we return some really large –ve value.
If n is of size 1 then just return a[0].
If n has size >=2…let us see an example.
```



1. maximum value is the larger of a[0] and the maximum in the range a[1..n-1].
2. This is computed by a recursive call: max_array(a+1,n-1).

```c
#include <stdio.h>
#define MAX_NEG -9999

int max( int a, int b);
int maxarray( int a[], int n)
{

        if(n==0)
                return MAX_NEG;

        if(n== 1)
                return a[0];

        return max( a[n-1], maxarray(a, n-1) );
}

int max( int a, int b);
{
        return( ( a>b)?a:b);
}

int main()
{
        int arr[]={100,10,4,20,45,56,72,43,33};
        printf("maxarray = %d\n",maxarray(arr, 9) );
        return 0;
}
```

Find the maximum of the the numbers in an array.

```
int max_array(int a[], int n) {
      int maxval;
      if (n == 0) return -99999; /* some large -ve number*/
      if (n==1) return a[0];      /* 1 element array */
      /* otherwise n >=2. */
      /* Find the largest element in the array a[1..n-1]  */
      maxval = max_array(a, n-1);
      return max(a[n-1],maxval);
}
```

How good is this program? Is it better, equal or worse than a standard iterative program we would write.

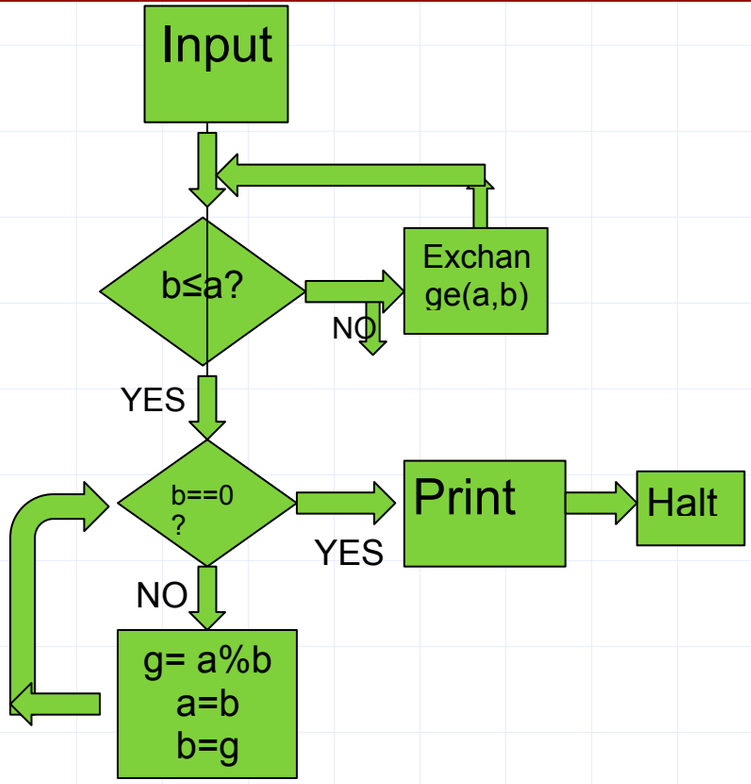
The questions are?

1. How much time does the recursive program take?
2. How much space (including stack depth) does it consume?

Example 4: GCD of two numbers

Find the gcd of two numbers.

```
int gcd(int a, int b);
Base case
   if b == 0, return a;
Reduction Step:
   return gcd (b, a%b);
```



```c
#include <stdio.h>

int gcd(int a, int b)
{
        if(b == 0)
                return a;
        else
                return gcd(b,a%b);
}

int main()
{
        int m,n;
        scanf("%d %d",&m,&n);
        if(m<n)
        {
                int tmp=m;
                m = n;
                n=tmp;
        }
        printf("gcd = %d\n",gcd(m,n) );
        return 0;
}
```

Coin Collection

You have an $n \times n$ grid with a certain number of coins in each cell of the grid. The grid cells are indexed by (i, j) where $0 \leq i, j \leq n - 1$.

For example, here is a 3x3 grid of coins:

	0	1	2
0	5	8	2
1	3	6	9
2	10	15	2

Coin Collection: Problem Statement

- You have to go from cell (0, 0) to (n-1, n-1).
- Whenever you pass through a cell, you collect all the coins in that cell.
- You can only move right or down from your current cell.

Goal: Collect the maximum number of coins.

Consider the example grid

5	8	2
3	6	9
10	15	2

There are many ways to go from (0,0) to (n-1,n-1)

5	8	2
3	6	9
10	15	2

Total = 35

5	8	2
3	6	9
10	15	2

Total = 25

5	8	2
3	6	9
10	15	2

Total = 31

5	8	2
3	6	9
10	15	2

Total = 30

5	8	2
3	6	9
10	15	2

Total = 23

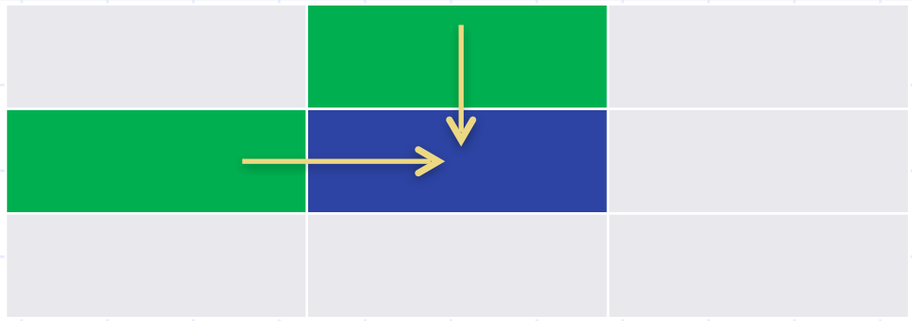
5	8	2
3	6	9
10	15	2

Total = 36

Max = 36

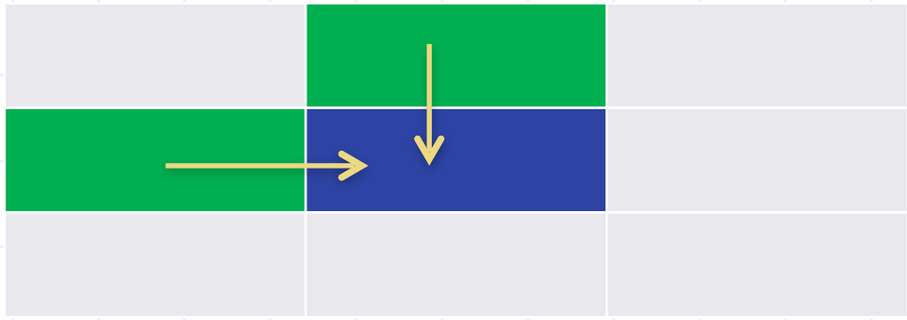
Solution Idea

- Consider a portion of some matrix



- What is the maximum number of coins that I can collect when I reach the blue cell?
  - This number depends only on the maximum number of coins that I can collect when I reach the two green cells.
  - Why? Because I can only come to the blue cell via one of the two green cells.

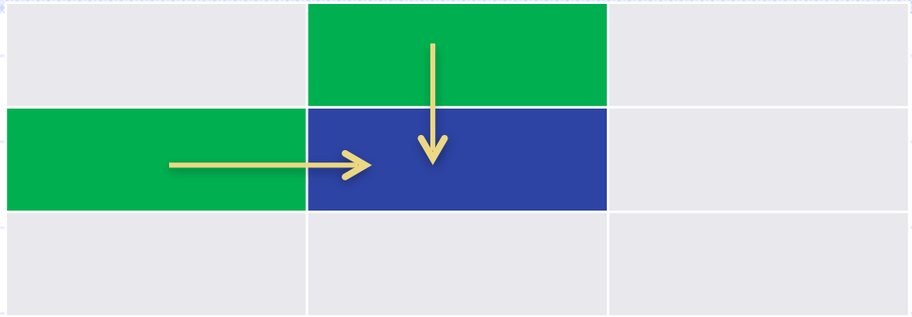
Solution Idea (dynamic programming)



Max-coins (bluecell) =

max(Max-coins (greencell1),
Max-coins (greencell2))
+ No. of coins (bluecell))

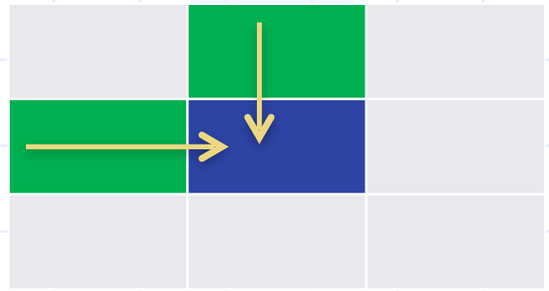
Solution Idea



- Let a(i,j) be the number of coins in cell(i,j)
- Let coin(i,j) be the maximum number of coins collected when travelling from (0,0) to (i,j).
- Then,

$$coin(i,j) = max(coin(i,j-1), coin(i-1,j)) + a(i,j)$$

Solution using Recursion



- Let a[i][j] be the number of coins in cell(i,j)
- Solution using recursion - coin_collect(a, i, j)

  if i == 0 && j==0, return a[i][j]

  //first row

  if i == 0, return a[i][j] + coin_collect(a,i, j-1)

  //first column

  if j == 0, return a[i][j] + coin_collect(a,i-1,j)

- Else

  return a[i][j]+ max(coin_collect(a,i,j-1), coin_collect(a,i-1,j))

```
#include <stdio.h>

int max(int a, int b);
int coin_collect(int m[][100], int i, int j);

int main(){
  int m[100][100],i,j,n;

  scanf("%d", &n);
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      scanf("%d", &m[i][j]);

  printf("%d\n", coin_collect(m,n-1, n-1));
  return 0;
}
~
```

```
int coin_collect(int m[][100], int i, int j)
{
   if(i == 0&&j == 0)
      return m[0][0];
   if( i == 0)
      return m[i][j] + coin_collect(m, i, j-1);
   if( j == 0)
      return m[i][j]+coin_collect(m, i-1, j);
   return m[i][j]+ max(coin_collect(m, i-1, j), coin_collect(m, i,
j-1) );
}
```

```
int max(int a, int b){
  if (a>b) return a;
  else return b;
}
```

Iterative vs Recursive

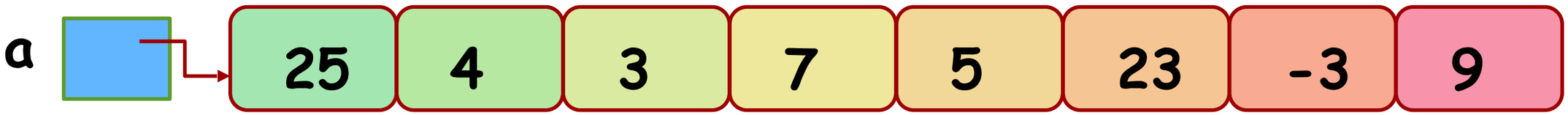
Which is better? the recursive formulation or the iterative formulation?

1. Logic is the same.
2. So the number of steps is of the same order.

Space used?

4. Iterative program uses space defined
5. Recursive program: Depends on the stack depth for the program?

Array's Maximum, once again



max_array(a,8)

calls max_array(a+1,7)

calls max_array(a+2,6)

calls max_array(a+3,5)

calls max_array(a+4,4)

calls max_array(a+5,3)

calls max_array(a+6,2)

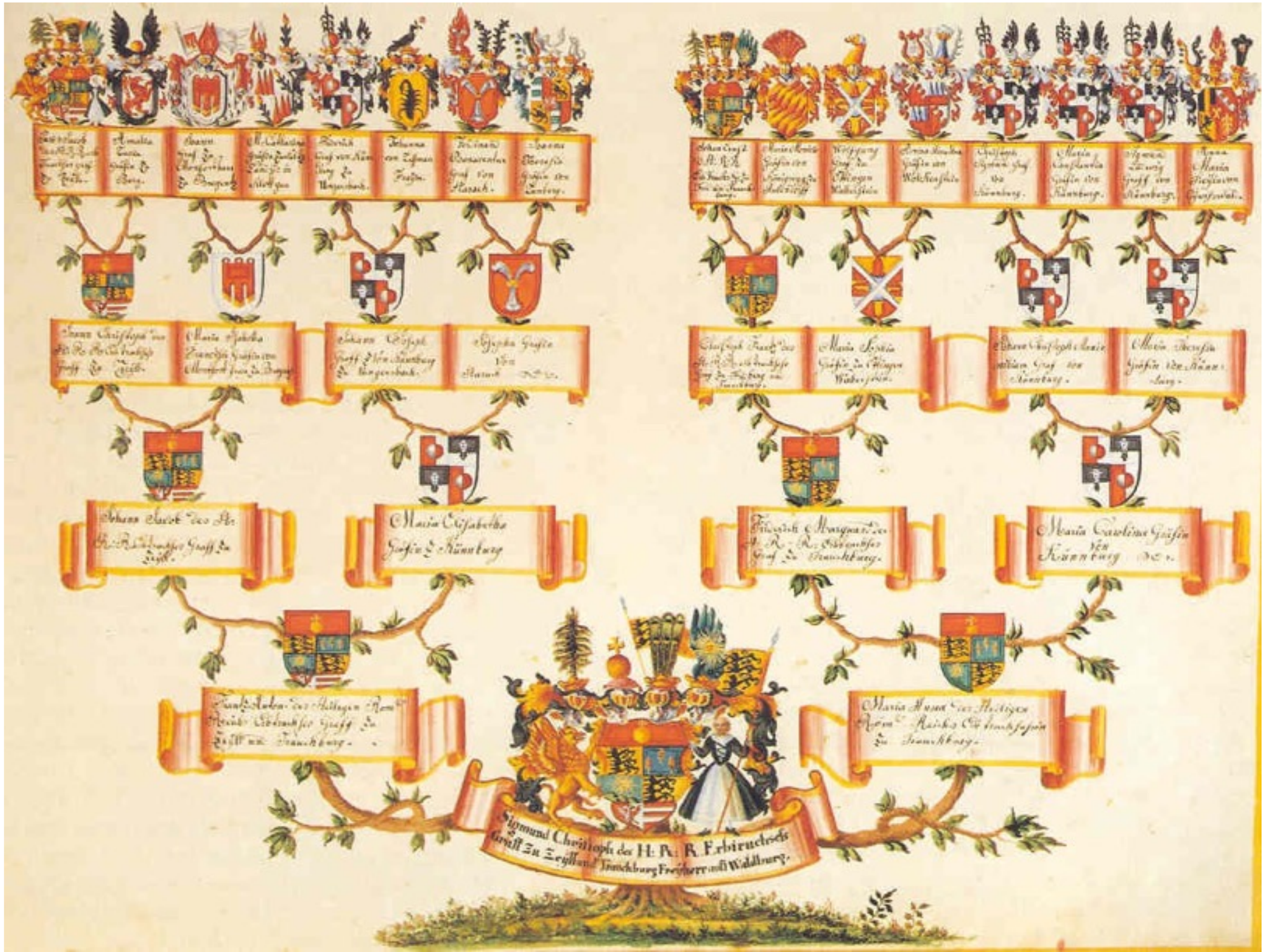
calls max_array(a+7,1)

```
int max_array(int a[], int n) {
    int maxval;
    if (n == 0) return -99999;
    if (n==1) return a[0];
    maxval = max_array(a, n-1);
    return max(a[n-1],maxval);
}
```

Stack Depth is n

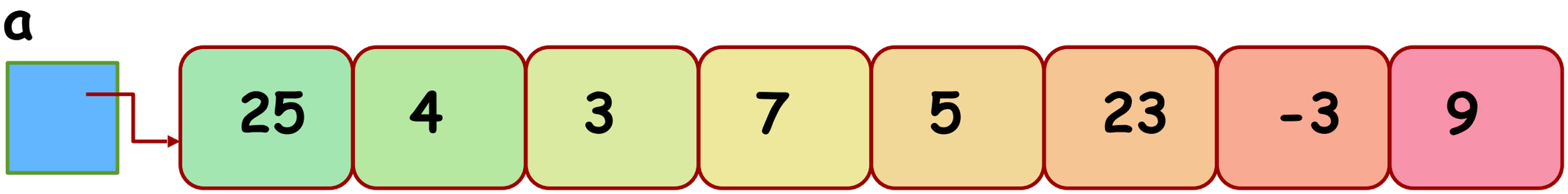
Can we reduce the stack depth?

Recursion II - Two-way Recursion

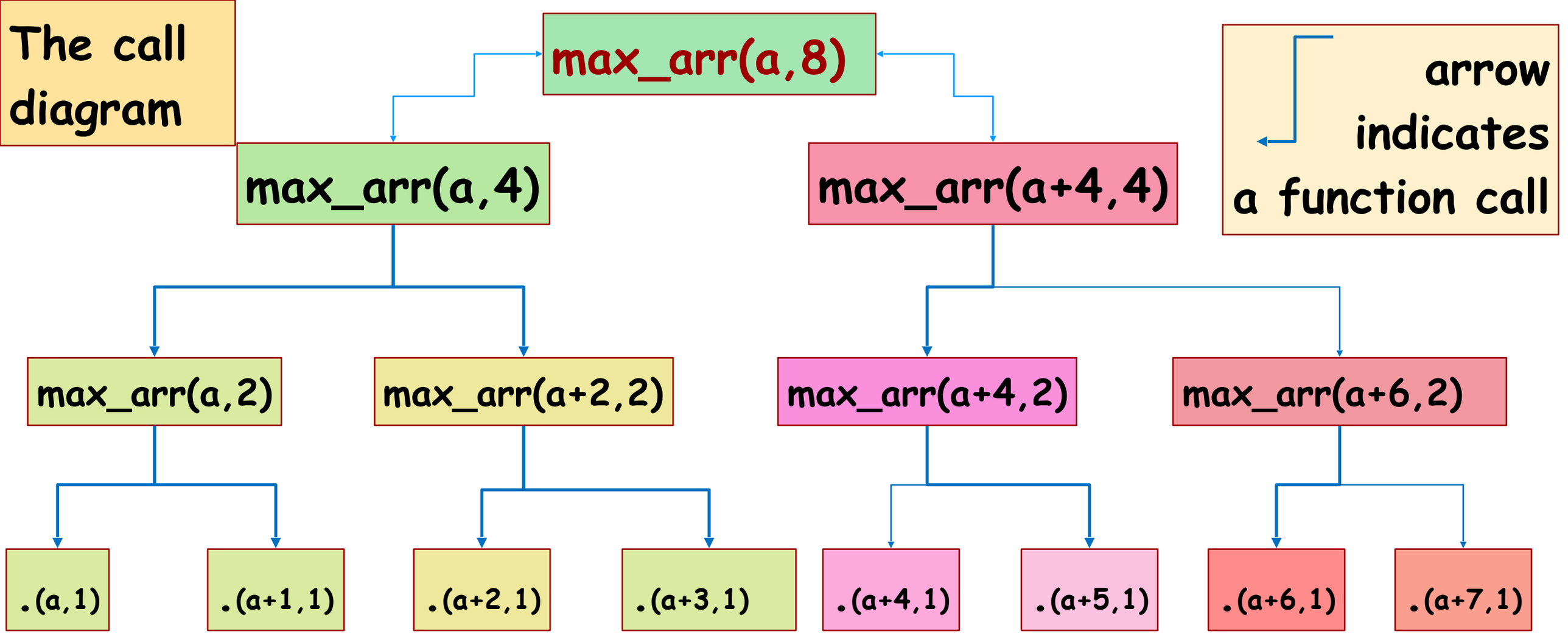


Can we reduce the stack depth?

1. Divide the array a into about two equal halves: a[s … e/2 -1] and a[e/2 … n-1].
2. Recursively find the maximum element in each half and return the larger of the two maxima.
3. As before: recursion exits when n is 0 or n is 1.
4. If n is 1 then return end element, if n is 0 return -INFTY.



The call diagram



Stack depth (length of the longest path in call stack) ≈ 1+log n.