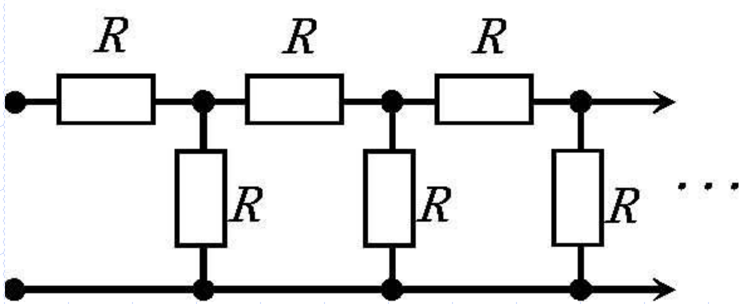
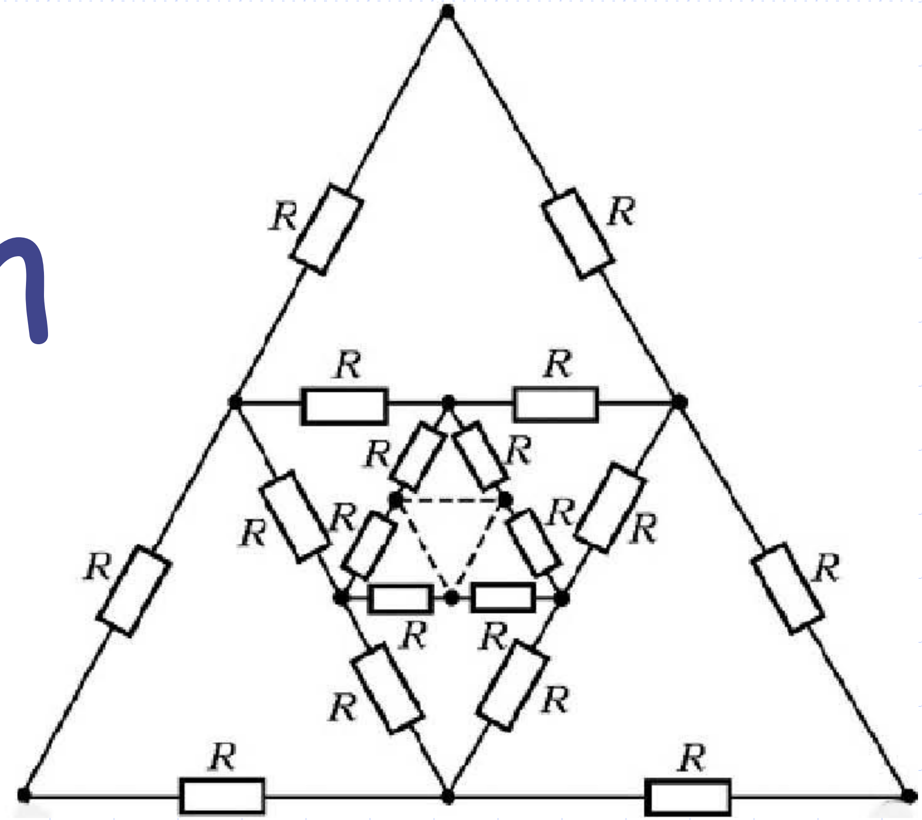
ESC101: Introduction to Computing

Recursion

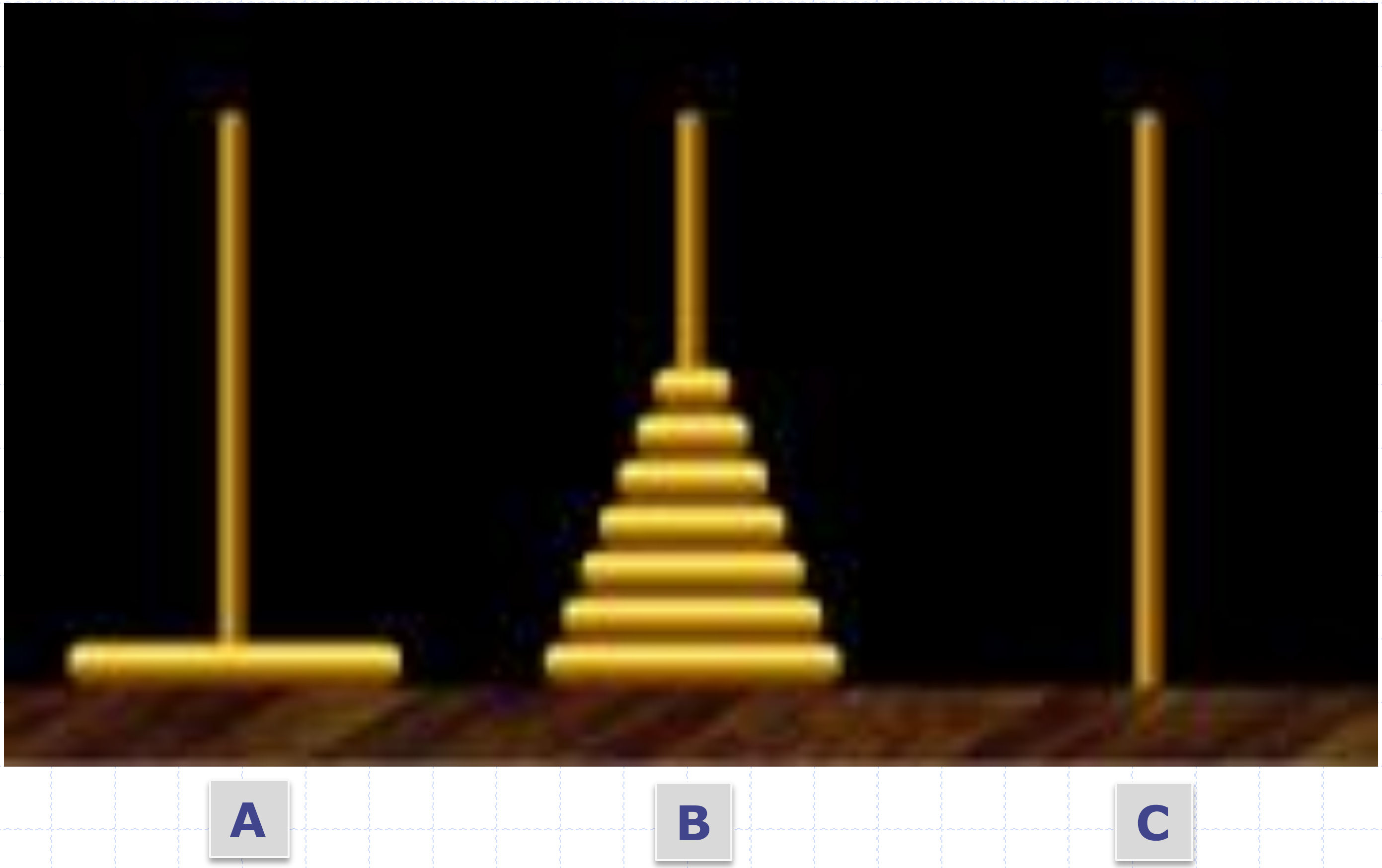




Recursion : Tower of Hanoi

A B C

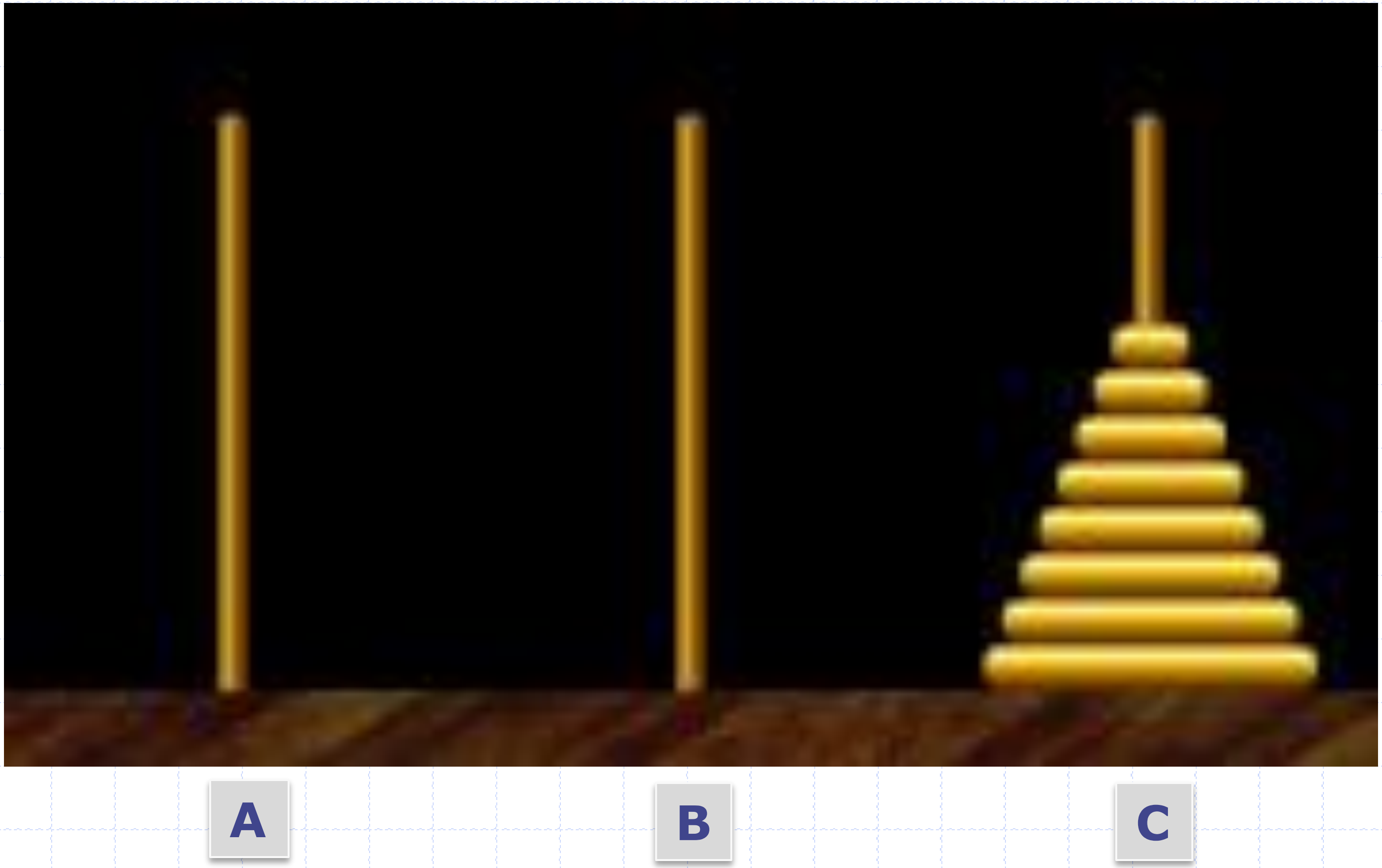
Recursion : Tower of Hanoi ..2



Recursion : Tower of Hanoi ..3

A B C

Recursion : Tower of Hanoi ..4



Recursion

- A function calling itself, *directly* or *indirectly*, is called a *recursive function.*
  - The phenomenon itself is called recursion
- Examples:
  - Factorial:

```
0! = 1
n! = n * (n-1)!
```

  - Even and Odd:

```
Even(n) = (n == 0) || Odd(n-1)
Odd(n)  = (n != 0) && Even(n-1)
```

Recursive Functions: Properties

- The arguments change between the recursive calls

  5! = 5 * 4! = 5 * 4 * 3! = …

- Change is towards a case for which solution is known (base case)
- There must be one or more base cases

  0! is 1

  Odd(0) is false

  Even(0) is true

Recursion and Induction

When programming recursively, think inductively

- Mathematical induction for the natural numbers
- Structural induction for other recursively-defined types (to be covered later!)

Recursion and Induction

When writing a recursive function,

- Write down a clear, concise *specification* of its behavior,
- Give an *inductive proof* that your code satisfies the specification.

Constructing Recursive functions: Examples

Write a function search(int a[], int n, key) that performs a sequential search of the array a[0..n-1] of int. Returns 1 if the key is found, otherwise returns -1.

How should we start? We have to think of the function search() in terms of search applied to a smaller array. Don't think in terms of loops…think recursion.

Here's a possibility ….

search(a,n,key)

Base case: If n is 0, then, return 0.

Otherwise: /* n > 0 */

1. compare last item, a[n-1], with key.
2. if a[n-1] == key, return 1.
3. search in array a, up to size n-1.
4. return the result of this "smaller" search.

a

search(a,10,3)

31	4	10	35	59	31	3	25	35	11

Either 3 is a[9]; or search(a,10,3) is same as the result of search for 3 in the array starting at a and of size 9.

```
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

Let us do a quick trace.

a

E.g., (0) search(a,5,10)

31	4	10	35	59

a[4] is 59, not 10. call search(a,4,10)

a

(1) search(a,4,10)

31	4	10	35	59

a[3] is 35, calls search(a,3,10)

a

(2) search(a,3,10)

31	4	10	35	59

a[2] is 10, return 1

```c
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

Let us do another quick trace.

a

E.g., (0) search(a,5,3)

31	4	10	35	59

a[4] is 59, not 3. call search(a,4,3)

a

(1) search(a,4,3)

31	4	10	35	59

a[3] is 35, calls search(a,3,3)

a

(2) search(a,3,3)

31	4	10	35	59

a[2] is 10, calls search(a,2,3)

a

(3) search(a,2,3)

31	4	10	35	59

a[1] is 4, calls search(a,1,3)

a

(4) search(a,1,3)

31	4	10	35	59

a[0] is 31, calls search(a,0,3)

31	4	10	35	59

a

(5) search(a,0,3) returns 0

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a+1,n-1,key);
5. }
```

a

search(a,5,3)

31	4	10
35	59	

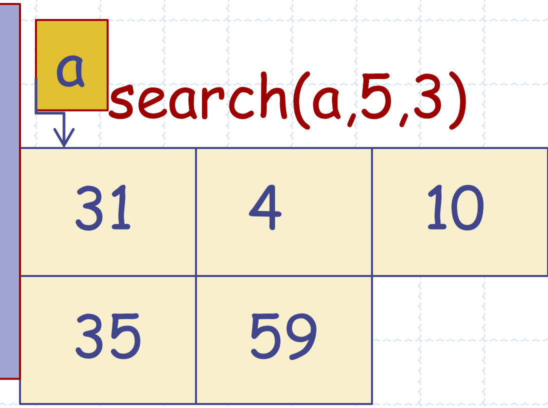
Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	
search(a,0,3)	search(a,1,3)	search.5	

recursion exits here

A state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a+1,n-1,key);
5. }
```



Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	
search(a,0,3)	search(a,1,3)	search.5	0

recursion exits here

A state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a+1,n-1,key);
5. }
```

a

search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	0

state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a+1,n-1,key);
5. }
```

a

search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	0

A state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a+1,n-1,key);
5. }
```

search(a,5,3)

a

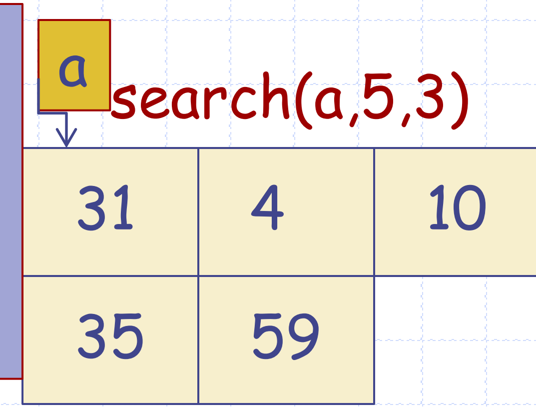
31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	0

A state of the stack

```
int search(int a[], int n, int key) {
    if (n==0) return ;
    if (a[0] == key)  return 1;
    return search(a+1,n-1,key);
}
```

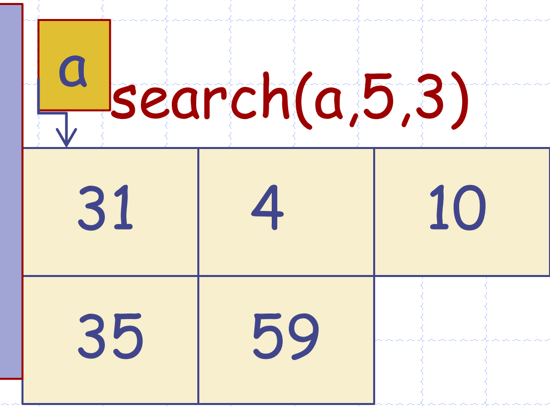


Stack

function	called by	return address	return value
search(a,5,3)	main()	---	0

A state of the stack

```c
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[0] == key)  return 1;
    return search(a+1,n-1,key);
}
```



search(a,5,3) returns 0. Recursion call stack terminates.

Searching in an Array

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

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

Disclaimer: Algorithm not tested for boundary cases

Searching in an Array

- One more recursive formulations
- Search2: search (a, start, end, key)
  - Search key between a[start]…a[end]

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
            || search(a, mid+1, end, key);
```

Disclaimer: Algorithm not tested for boundary cases

Estimating the Time taken

- Two types of operations
  - Function calls
  - Other operations (call them simple operations)
- Assume each simple operation takes fixed amount of time (1 unit) to execute
  - Really a very crude assumption, but will simplify calculations
- Time taken by a function call is proportional to the number of operations performed by the call before returning.

Estimating the Time taken

- Search1

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

  - Let T(n) denote the time taken by search on an array of size n.
  - Line 1 takes 1 unit (or 2 units if you consider if check and return as two operations)
  - Line 2 takes 1 unit (or 3 units if you consider if check, array access and return as three operations)
  - But what about line 3?

Estimating the Time taken

- Search1

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

  - What about line 3?
  - Remember the assumption: *Let T(n) denote the time taken by search on an array of size n.*
  - Line 3 is searching in n-1 sized array => takes T(n-1) units
  - But what about the value of T(n) ?

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - But what about the value of T(n) ?
  - Looking at the body of search, and the information we gathered on previous slides, we can come up with a recurrence relation:

$$T(n) = T(n-1) + C$$

  - We need to solve the recurrence to get the estimate of time

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - Solution to the recurrence?

$$T(n) = T(n-1) + C$$

$$T(n) = Cn$$

  - The worst case run time of Search1 is proportional to the size of array
    - Bigger the array, slower the search
  - What is the best case run time?
  - Which one is more important to consider?

Estimating the Time taken

- Search2
  - Recurrence?

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
          || search(a, mid+1, end, key);
```

$$T(n) = T(n/2) + T(n/2) + C$$

$$T(n) \propto n$$

  - The **worst case** run time of Search2 is also proportional to the size of array
    - Can we do better?

Binary Search for Sorted Arrays

Classic Recursive Algorithms

- Sorting
  - Quicksort, Mergesort, …
- Searching
  - Binary Search
- Traversals
  - (Tree) Inorder, Preorder, Postorder
- …

Around Easter 1961, a course on ALGOL 60 was offered … It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who **included recursion in their language** and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

- **The Emperor's Old Clothes, C. A. R. Hoare, ACM Turing Award Lecture, 1980**

Recursion vs Iteration

```
int fib(int n)
{
   int first = 0, second = 1, next, c;
   if (n <= 1)
      return n;
   for ( c = 1; c < n ; c++ ) {
      next = first + second;
      first = second;
      second = next;
    }
    return next;
}
```

```
int fib(int n)
{
     if ( n == 0 )
          return 0;
     else if ( n == 1 )
          return 1;
     else
          return fib(n-1)
                     + fib(n-2);
  }
```

fib(5)

fib(4) + fib(3)

fib(3) + fib(2) + fib(2) + fib(1)

fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0)

fib(1) + fib(0)