Recursion

Recursion

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

  0! = 1

  n! = n * (n-1) * (n-2)!

  Even(n) = (n==0) || Odd(n-1)

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

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! = 1

  or

  Odd(0) is false

  Even(0) is true

Recursion and Induction

- When programming recursively, think inductively.

How to prove:

f(n) = 1+2+3+4+...+n = $\frac{1}{2}$*n*(n+1)

We begin by checking if f(1) is true. Next we assume f(n) is true. Finally, we need to prove that f(n+1) is true.

Proof

f(n) = 1+2+3+4+...+n = $\frac{1}{2}$*n*(n+1)

f(1) = 1 = ½ * 1 * 2 = 1. Thus, f(1) is true.

f(2) = 1 + 2 = 3   (1/2)*2*3 = 3. Thus, f(2) is true.

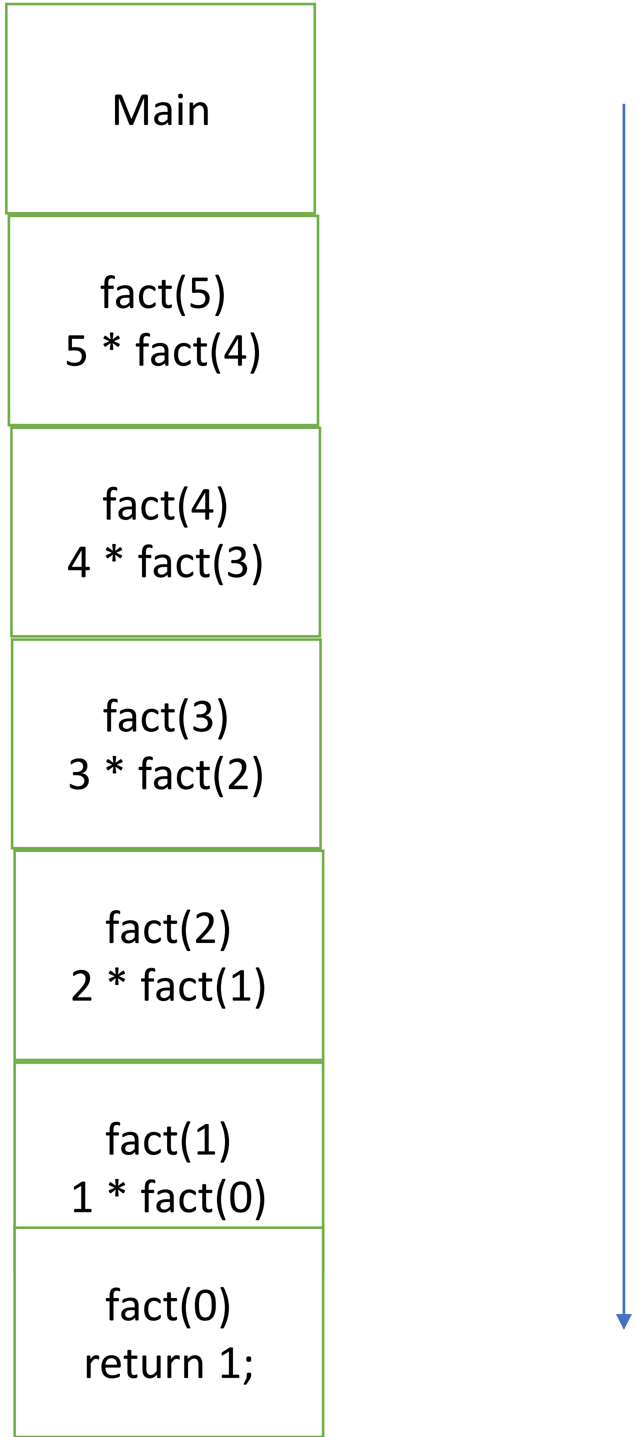
...

Assume f(n) = 1+2+3...+n = ½*n*(n+1) is true.

To prove f(n+1) is true.

1+2+3+...+n+(n+1) = ½ * n * (n+1) + (n+1)

f(n+1) = 1+2+3+...+(n+1) =1/2*(n+1)*(n+2). Hence, for all n, f(n) is true.


Main
fact(5)
5 * fact(4)
fact(4)
4 * fact(3)
fact(3)
3 * fact(2)
fact(2)
2 * fact(1)
fact(1)
1 * fact(0)
fact(0)
return 1;

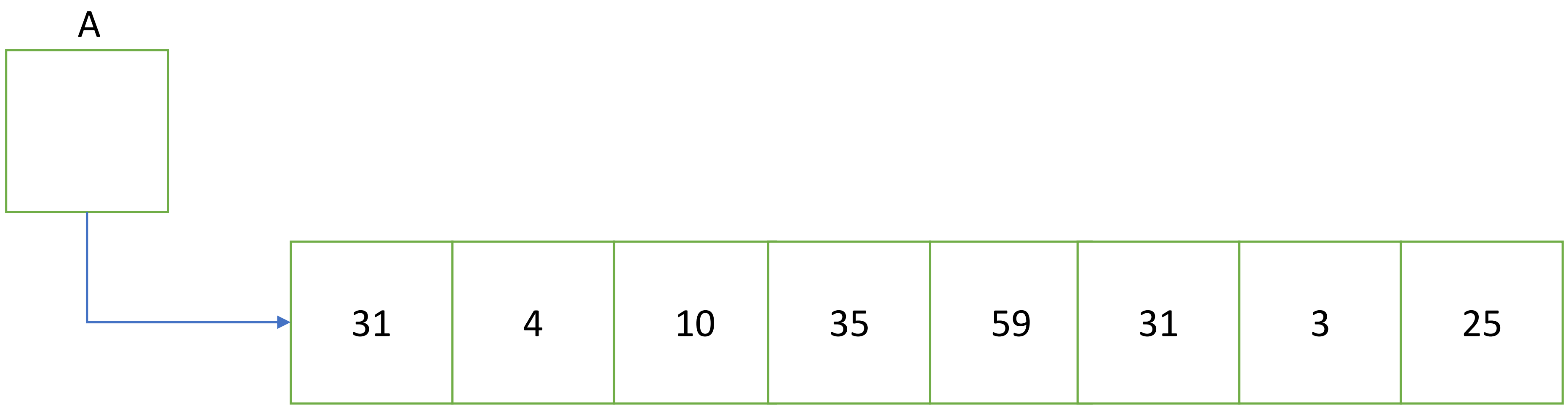
Example

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

- The think about the solution, think about searching an element in a smaller array. Don’t think in terms of loop...think in terms of recursion.

Solution

- Base case: If there are no elements, you can return 0.
- Otherwise:
- compare last item, a[n-1] with key.
- if a[n-1] == key, return 1
- else search key in the remaining array of size n-1 and return the result of this “smaller” problem.



search(A, 8, 10)

Either A[7] == 10 or search(A, 7, 10)

80 it does not exist

Program

```
#include<stdio.h>
int search(int a[], int nEle, int key) {
	if(nEle == 0) return -1;
	if(a[nEle-1] == key) return 1;
	return search(a, n-1, key);
}
void main() {
	int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
	int key;
	printf(“\nEnter a key to search:”);
	scanf(“%d”, &key);
	int pos = search(arr, 9, key);
	if(pos)
		printf(“Element found at position: %d”, pos + 1);
	else
		printf(“Element not found.“);
}
```

Recursion: Time Analysis

- Let us try to compute the time taken by the function “search”.
- Let us assume T(n) be the time it takes to search an element “key” in an array of size “n.”
- Assume that if condition takes 1 time unit to execute.

Thus, $T(n) = 1 + 1 + T(n-1)$

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

- The last statement is known as recurrence relation.

Solution to Recurrence Relation

- We know the following:

$T(n) = T(n-1) + C, \ T(0) = C$

$T(n-1) = T(n-2) + C$

$T(n-2) = T(n-3) + C$

...

$T(2) = 2C + C = 3C$

$T(1) = 2C$

Adding all, you get

$T(n) = C + nC \Rightarrow T(n)$ is proportional to n

Binary Search

- Can we search faster than "n" (linear search), where "n" is number of elements?
- Yes, if the elements are sorted in ascending or descending order.

Ex:	1	2	3	4	5	6	7	8	9
Ex:	9	8	7	6	5	4	3	2	1

To search: 7

How it works?

Elements	1	2	3	4	5	6	7	8	9
Index	0	1	2	3	4	5	6	7	8

To search: 7

Left = 0, Right = 8, Value at pos 4 != 7. Also, Value at pMid = (0+8)/2 = 4

os 4 < 7. Thus, Left = Mid + 1

Left = 5, Right = 8, Mid = (5+8)/2 = 6

Value at pos 6 == 7, Thus, element is found.

Methodology

Ex: 1 2 3 4 5 6 7 8 9

To search: 7

- Let us consider 3 variables: left, right, and middle.
- Initially, left = 0 (position in array), right= length.
- Calculate mid as (left+right)/2 and check if the key to be searched is found or greater or smaller.
- If the key is greater than mid, you need to search in right array. Else, you need to search in the left array.

Function

```
int binarysearch(int a[], int start, int end, int key) {
        if(start > end)
                return -1;
        int mid = (start + end)/2;
        if(a[mid] == key)
                return mid;
        else if(a[mid] > key)
                return binarysearch(a, start, mid – 1, key);
        else
                 return binarysearch(a, mid + 1, end, key);
}
```

Time Taken?

- Recurrence Relation

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

let n = 2^k

T(2^k) = T(2^k-1) + C

T(2^k-1) = T(2^k-2) + C

...

T(2) = T(1) + C

=> T(2^k) is proportional to k => T(n) is proportional to $\log_2 n$

Fibonacci Numbers

- Sequence like following

0 1 1 2 3 5 8 13

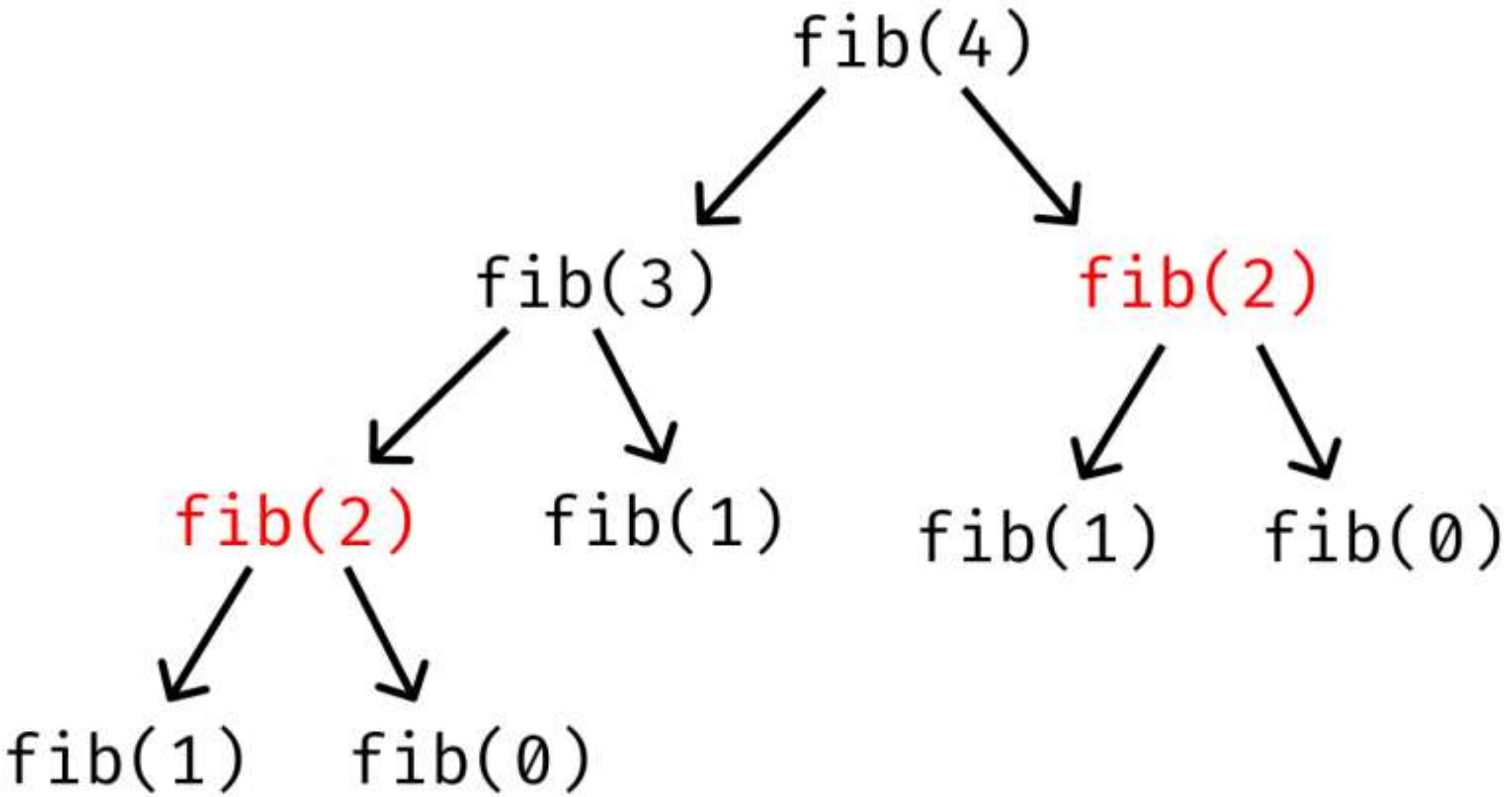
In general

$$F_n = F_{n-1} + F_{n-2}$$

Code

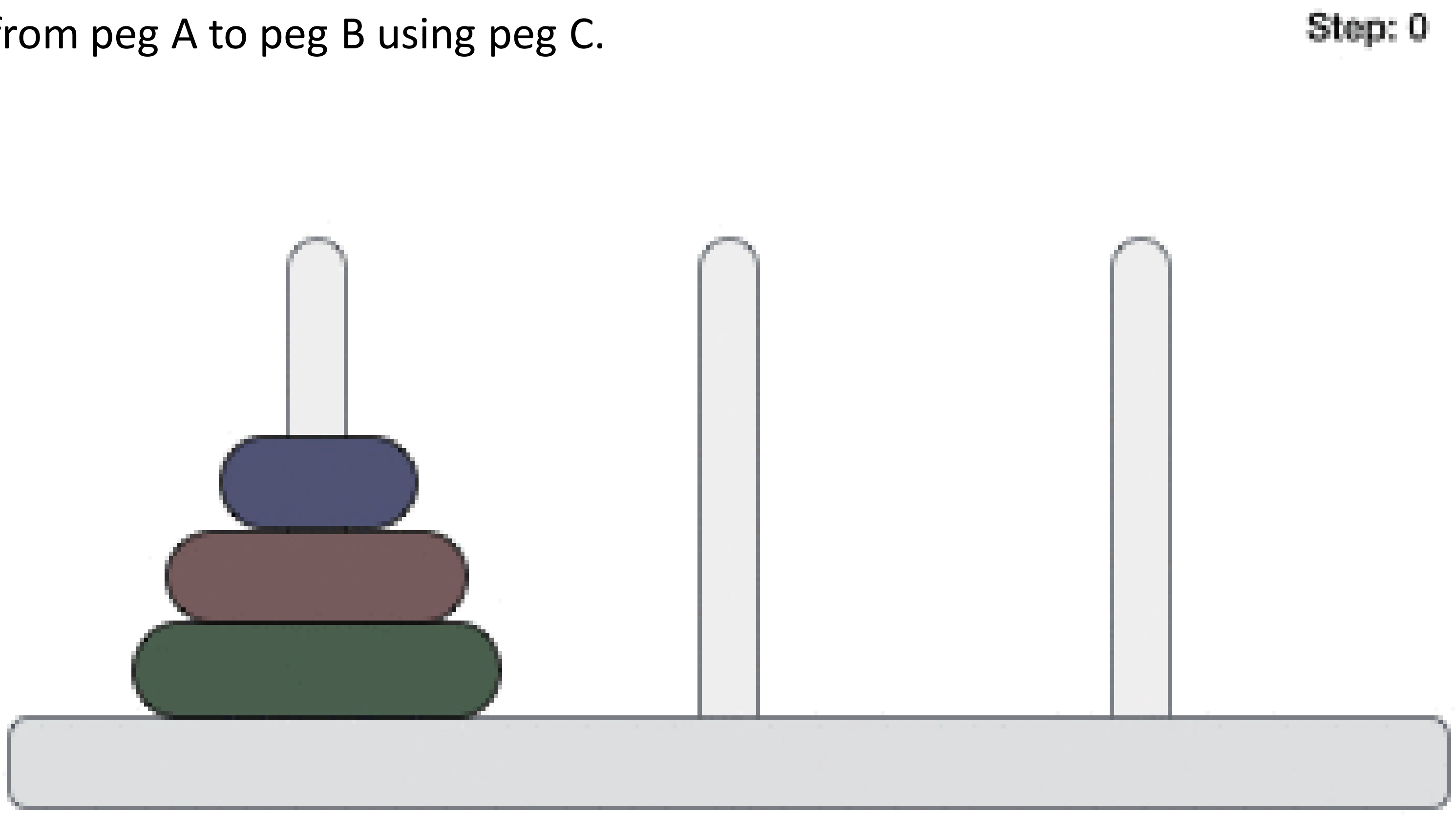
```
int fibo(int term) {
	if(term <= 1)
		return term;
	return fibo(term - 1) + fibo(term - 2);
}
```

Calculation



Tower of Hanoi

Move all discs from peg A to peg B using peg C.

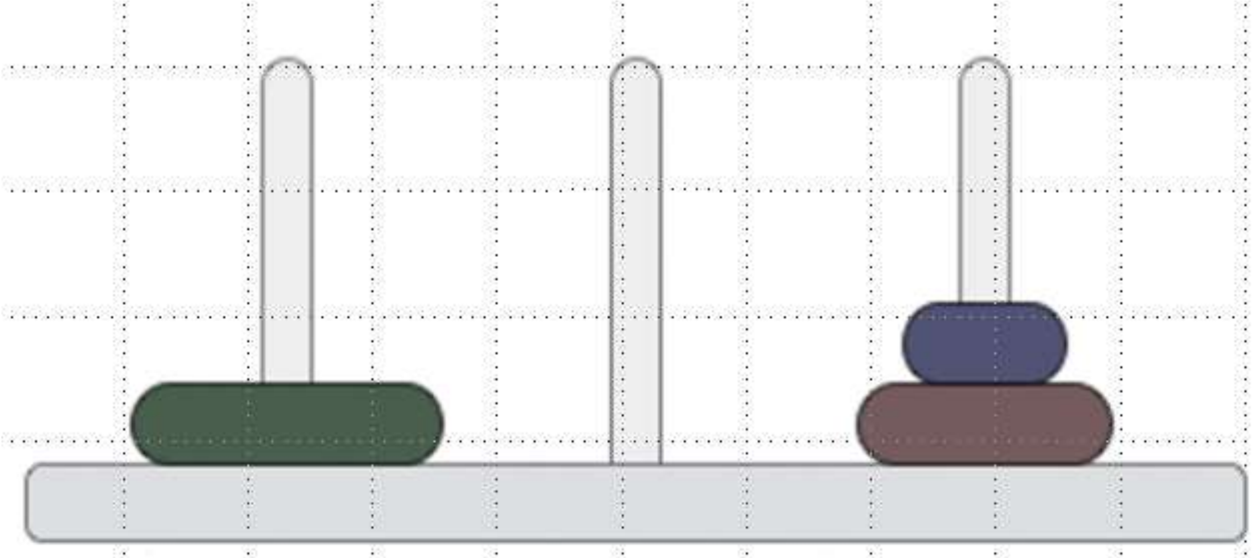


Rules

- Only one disk can be moved at a time
- No disk may be placed on top of a smaller disk

How to think about it recursively?

Lets say if you have the following situation, then you need to move disk from peg A to peg B, then you need to move remaining disk from peg C to B.



Solution

- So, if you have a function to move n disks from peg X to peg Y using peg Z, you can use it again to solve the situation.

Code

```
void Hanoi(int n, char A, char B, char C) {
        if(n==0) return;
        Hanoi(n-1, A, C, B);
        printf("Move 1 disk from %c to %c", A, B);
        Hanoi(n-1, C, B, A);
}
```

Summary

- Advantage
  - Elegant Solution
  - Fewer Variables
  - Easy to implement once you figure out the recursive definition.
- Disadvantage
  - Debugging is difficult
  - Figuring out the logic is sometimes difficult
  - Can be inefficient