# Designing recursive algorithms/functions

- 1. Base case
- 2. Making progress
- 3. Design rule
- 4. No compund interest

```
void printOut(int n)
  if (n < 10)
     printDigit(n);
     return;
   printOut(n/10);
   printDigit(n%10);
```

```
void printOut(int n)
                                      Output?
  if (n < 10)
     printDigit(n);
     return;
   printDigit(n%10);
   printOut(n/10);
```

# Multiply a value by an integer

```
double mul(double val, int n)
   if (n==1)
      return val;
   double t = mul(val, n-1);
   return val + t:
```

```
axb is
a+a+a+a+...+a
b times
```

### Integral power of a number

```
double power (double num, int p)
  if (p==0)
     return 1;
  double t = power(num, p-1);
  return num * t;
```

### Return product of all values in an array

```
double prod(float a[], int size)
   if (size==0)
                                       what's wrong here?
      return <del>0.0</del>; C
   double t = prod(a, size-1);
   return t * a[size-1];
```

## Maximun/minimun value in a array

```
double max(double nums[], int count)
  if (count==1)
     return nums[0];
  double t = max(nums, count-1);
  double lv = a[count-1];
  return ly > t? ly: t:
```

# Summing up

There must be some n/size/count as argument of the recursive function.

That should make progress in recursive calls towards the base case.

Base case(s) are for its very small values; typically 0 or 1. In Base case(s), answer should be direct (without any computation).

```
Integral power of a number (version 2)
   double power (double num, int p)
      if (p==0)
         return 1;
                                                      is int division
      double t1 = power(num, p/2);
      double t2 = power(num, p-p/2);
      return t1 * t2;
```

## Integral power of a number (version 3)

```
no year sive function
double power (double num, int p)
  if (p==0)
     return 1;
  double v = power(num, p/2);
                                    benifit?
  v = v * v:
  return (p%2==0) ? v : v*num;
```

#### n'th Fibo number

```
int fibo(int n)
{
    if (n==1 || n==2)
    {
        return 1;
    }
    return fibo(n-1) + fibo(n-2);
}
```

## n'th Fibo number (efficient logic)

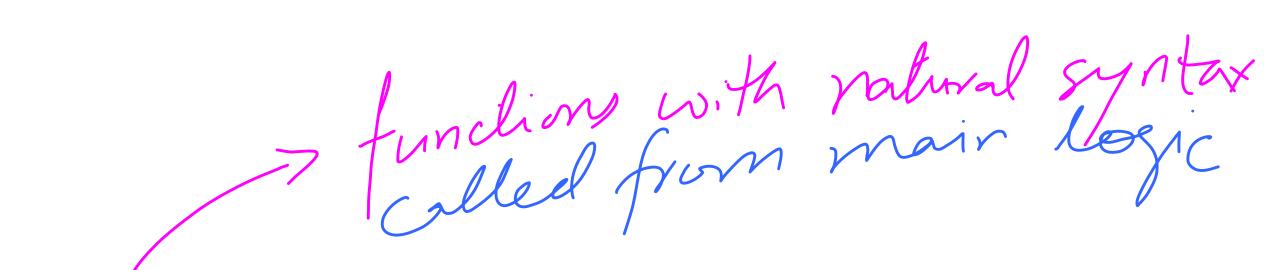
```
double f(int n, int a[])
   if (a[n-1] == 0) {
       if (n==1 || n==2)
          a[n-1] = 1;
       else
          a[n-1] = f(n-1, a) + f(n-2, a);
   return a[n-1];
```

```
how to call
```

```
cout \ll fibo(15);
int fibo(int n)
   call to the yechrine
  return f(n, tmp);
```

using logic tricks, algo's can be made efficient

formal theory will be discussed later in the course



Wrapper functions

VS

Helper or auxiliary functions

yearsive functions with extra

extra

parameters

extra

Parameters

Called from wrappers

#### <u>Linear search</u>

```
bool search (double nums[], int size, double val)
  if (size==0)
     return false;
  return search(nums, size-1, val) | (nums[size-1] == val);
  // return (nums[size-1] == val) || search(nums, size-1, val);
```

#### Linear search

```
int position (double nums[], int size, double val)
   if (size==0)
      return -1; // better to throw exception
   if (nums[size-1] == val)
      return size-1;
   else
      return position(nums, size-1, val);
```

## Binary search

```
int bsearch(double nums[], int li, int hi, double val)
    if (li > hi)
        return -1; // better to throw exception
    mi = (li + hi) / 2;
    if (nums[mi] == val)
        return mi;
    else if (val < nums[mi])
        return bsearch(nums, li, mi, val);
    else if (val < nums[mi])
        return bsearch(nums, mi, hi, val);
```

```
// call an overloaded function
p = bsearch(a, n, x);
// which should call the left function
```

#### **Bubble Sort (recursive)**

```
void sort(int a[], int n)
  if (size==1) // or 0?
     return;
  sort(a, n-1);
  bubble(a, n); // bubble up the n'th value in array
```