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);
}
```

assuming, can print a digit from 0 to 9

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

Output ?

assuming can print a digit from 0 to 9

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;
}
```

$a * b$ is

$$\underbrace{a+a+a+a+\ldots+a}_{b \text{ times}}$$

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;
}
```

b^p is

$\underbrace{b * b * b * \cdots * b}_{p \text{ times}}$

Return product of all values in an array

```
double prod(float a[], int size)
{
    if (size==0) 1
    {
        return 0.0; a[0]
    }
    double t = prod(a, size-1);
    return t * a[size-1];
}
```

what's wrong here ?

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 lv > t ? lv : 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;
    }
    double t1 = power(num, p/2);
    double t2 = power(num, p-p/2);
    return t1 * t2;
```

$$b^p = b^{\frac{p}{2}} \cdot b^{p-\frac{p}{2}}$$

$\frac{p}{2}$ is int division

Integral power of a number (version 3)

```
double power (double num, int p)
{
    if (p==0)
    {
        return 1;
    }
    double v = power(num, p/2);
    v = v * v;
    return (p%2==0) ? v : v*num;
```

no real need of other recursive function call.

benifit ?

when p is odd

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 << fibo(15);
```

natural call

```
int fibo(int n)
{
    int tmp[n] = { 0, 0, 0, ... } ;
    return f(n, tmp);
}
```

call to the recursive function with extra parameter using logic tricks, algo's can be made efficient

formal theory will be
discussed later in the course

Wrapper functions

functions with natural syntax
called from main logic

vs

Helper or auxilary functions

recursive functions with extra
extra parameters
called from wrappers

Linear search

```
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
}
```

NOT DISCUSSED in CLASS