Recursion

This is a relatively important part in your Matlab studying, and it might be a little bit complex to understand. Recursion is something that occurs frequently in algorithm design, so I hope you understand this part as soon as possible.

What is Recursion

Recursion is a self-calling behaviour of a function. In the implementation of a certain function, you can let it use itself to solve complex problems.

The most famous example of recursion must be **Fibonacci Sequence**. Suppose the first two elements of a sequence are all 1 and the other elements follows the rule that$a_n = a_{n-1} + a_{n-2} (n \geq 3)$ What is the value of a_{n_0} for a certain integer n_0?

Of course the general formula can be calculated, but in programming we often use the following methods.

1. Find the values of a_{n_0-1} and a_{n_0-2} using this method.
2. Add the two values together.

However it is obvious that this method won't work, as the first step will be infinite. But keep in mind that we know the values of a_1 and a_2, so we can let the first step stop at the two situations. So the algorithm should be revised like the following to find a_{n_0}.

1. If n_0 is exactly 1 or 2, then the value is just 1, and the following two steps can be skipped.
2. Find the values of a_{n_0-1} and a_{n_0-2} using this method.
3. Add the two values together.

And to implement this algorithm in Matlab, it can be written like this:

```
function A = Fibonacci(n)
    if (n==1)||(n==2)
      A = 1;
    else A = Fibonacci(n-1) + Fibonacci(n-2);
    end
end
```

And I believe that you can understand the mechanism of this function. Recursion is exactly the process that a function calls itself until it finds a boundary condition(in this example it is a_1 and a_2). And then the function start to get back to the original n from the boundary conditions.

Some Tips about Recursion

1. Get clear about the recursive relationship between functions as it is not so simple as that in Fibonacci Sequence.
2. Set a resonable boundary condition. Usually the boundary condition is not just the "starting point". In many cases you should define the **invalid input argument** as a special boundary case as it might appear in the self-calling process.
3. The basic requirement of boundary condition defining is that all self-calling process can stop at these conditions.

Divide and Conquer

In fact such kind of usage of recursion has another name called divide and conquer.

The main idea of divide and conquer is to split a problem into smaller problems which are similar with the original problem but with less complexity.

In the Fibonacci example, the problem of finding a_{n_0} is split into problems of finding a_{n_0-1} and a_{n_0-2}, which are similar to the original problem, but is simpler since the index is smaller.

Another crucial usage of the idea of divide and conquer is in the sorting algorithms. Here is the implementation of one special sorting algorithm called Merge Sort . I believe that this example can better help you understand the concept of divide and conquer. You don't have to understand the code as it is written in C++ but you should know how it works.

Recursion for Non-Math Problems

Sometimes the problem given might not be a pure-mathematic problem as finding a number in a sequence, and in many times the problem will be more realistic and complex.

However, the main idea of using divide and conquer is still the same in these situations. And I think the crucial part is to decide the form of the function. In another word, the input argument and the return value of the function.

As for the return value, you can just set it as the final result as you want.

In the part of deciding the input arguments. My suggestion is that your settlement of input arguments should be able to completely describe a "state", no matter it is the final state or just an intermidiate state. Here I will introduce an example called **Backpack Problem**.

> Suppose you are a skilled thief and you sneaked into a museum to steal something. There are many exhibits in the museum and you should bring some back.
> Of course you want the total value of what you steal be as large as possible. However, you cannot bring everything home as the total weight will be too large to carry. So it is time for you to choose some to put into your backpack.
> Suppose that there are n exhibits. You know exactly the value a_i and the weight w_i of the i_{th} exhibit and you know that the maximum total weight that you can carry is $maxw$. The question is, what is the maximum total value of what you can bring back?

In this problem, I will use these things to define a "state":

1. The weight that I can still put into my backpack.
2. The number of exhibits that I can choose.

To be more clear my function represents "**the maximum total value if I choose from the first k exhibits with maximum weight v**" and what we want to solve is the situation where $k = n$ and $v = maxw$.

So the function looks like:

```
function Y = Backpack(k,v)
...
end
```

And we want to find $Backpack(n, maxw)$.

Now think about the k_{th} exhibit. Actually I can choose whether or not to bring it.

If I choose to bring the k_{th} exhibit, I have only $v - w_k$ of weight to put the rest $k - 1$ exhibits, thus the maximum total value is $Backpack(k-1, v-w_k)$. But since I already take the k_{th} one, the value should be added with a_k, and it will be $a_k + Backpack(k-1, v-w_k)$.

If I don't take the k_{th} exhibit, then I have v weight left to choose from the rest $k - 1$ exhibits, and the maximum total value is $Backpack(k-1, v)$.

And it is obvious that the maximum value of choosing from the first k goods with v maximum weight is the larger one of the former two situations. Thus the recursive expression can be represented as $Backpack(k, v) = max\{a_k + Backpack(k-1, v-w_k), Backpack(k-1, v)\}$

Now it is time to decide the boundary conditions. As there are two input arguments, there might be two different kind of boundary conditions:

1. If $k = 0$, that means that I should choose from nothing, then the total value will be of course 0.
2. If $v \leq 0$(here it is possible that $v < 0$), then I cannot carry anything, so the value will also be 0.

As the boundary condtitions are all settled, the function can be implemented(Suppose vector A and W store the data of a_i and w_i):

```
function Y = Backpack(k,v)
  if (k==0)||(v<=0)
    Y = 0;
  else
    Y = max(A(k)+Backpack(k-1,v-W(k)),Backpack(k-1,v));
  end
end
```

Then you just need to calculate $Backpack(n, maxw)$

This example is a rather complex one, so if you can understand and describe/implement the algorithm by yourself, you have actually got the hang of non-math recursion.

There will be some more examples such as the famous **Hanoi Tower** problem and **8-Queen** problem, and these will be mentioned in RC classes.

P.S.

In this chapter I also leave some questions for you in the "practice" directory.

Another tip is that **recursion is not a method to save running time and space of a program**. It is because that the process of calling a function takes a lot of time and space. The reason for using recursion is that it is a more undestandable and implementable way for programmers and code readers.