This lecture continues the discussion about recursion.

This lecture will explain the integer partition in details.

More specifically, this lecture explains how to write a program that prints all possible ways to partition a positive integer.

As a review, the integer partition problem takes a positive integer and breaks it into the sum of one or several positive integers. The original number itself is also allowed. For example, two can be broken into one plus one or two itself. Number three can be divided into one plus one plus one, or one plus two, or two plus one, or three itself.

Four can be divided into one plus one plus one plus one, or one plus one plus two, or one plus two plus one, or one plus three, or two plus one plus one, or two plus two, or three plus one, or four itself.

The problem we want to solve is to write a program that prints these partitions.

Let’s look at the output more closely. Consider the scenario when we want to partition number 4. Let’s focus on the first number used for the partitions.

The first number can be one, or two, or three, or four. This suggests using a for loop with numbers one, two, three and four.

Next, let’s inspect everything after the first number. When the first number is one, three is left and we need to partition three. When the first number is two. Two is left and we need to partition two.

Thus, the method we are going to use has these steps: Suppose we want to partition number N. .

Step number one: if the number to partition is zero, stop.

Step number two: select a number from one, two, three, all the way to the number we want to partition.

Step number three: partition the remaining number, N minus one, N minus two, down to zero.

You can find the partition program at this git hub site.

This program has three functions. The first function is the print partition function. It takes an array and an integer as the length of the array.

This function prints the plus sign between the numbers in the array.

The second function is the main function. It takes one integer from A R G V one. This is the number to be partitioned. The main function allocates heap memory as large as the number, calls the partition function with three arguments: the array to store the used numbers, zero, and the number to be partitioned. Then, the main function frees the memory and returns exit success.

Next, we are going to study the partition function carefully. We will see how the stack memory changes as this recursive function progresses.

For simplicity, I will not mark the return location. This function has no return value. Thus, there is no value address.

The partition function is actually quite short. At the top, it checks the stop condition. If the number to be partitioned is zero. The function prints the array and returns. Otherwise, the function uses a loop to go through one, two, three, to the value to be partitioned. Inside this loop, there are only two lines. The first line assigned the value of the loop. The second line calls the function again with the array as the first argument. The second argument is the index plus one. The third argument is the number to be partitioned minus the loop value.

Please take a moment to inspect this function.

The return statement inside the stop condition is unnecessary. The reason is that when the third argument is zero, this function will not enter the for loop. This loop value starts at one and can be as large as the value of the third argument. If the third argument is zero, the function will not enter the for loop and as a result this function call ends.

Even though the return statement is unnecessary, it is added there for clarity.

Please remember that the main function calls this partition function with the heap memory, zero, and the number to be partitioned as the three arguments.

Let’s understand how this function works by tracing the changes in the stack and heap memory.

When the partitioned is called the first time, all values in the heap memory are uninitialized and thus marked U. .

The first argument is the address of the heap memory.

The second argument is zero.

The third argument is the value to be partitioned and it is four.

The local variable V A L . is not initialized yet and also marked U . .

Since the third argument is not zero, the stop condition is false.

The program moves to the for loop and initialized V A L . to one.

The next line assigns one to the first element of the array. The argument A R R . stores the address of the heap memory and it is 10000. The index is zero and this means the first element of the array. V A L starts at one.

Thus, one is assigned to the value of the first array element.

The next line is a function call. The first argument is unchanged and it is the address in the heap memory, 10000.

The second argument is I N D plus one and it is one.

The third argument is left minus V A L . Left is four and val is one. Thus, the third argument is three.

The program enters the function to check the stop condition. The value of left is three and the stop condition is not met.

The function moves to the for loop and initializes val to one.

The next line assigns one to the second element of the array.

The next line is another function call.

The first argument is still the address of the heap memory. The second argument is I N D plus one and it is two. The third value is left minus val and it is two.

The function first checks whether the stop condition is met. Since the third argument is two, not zero, the stop condition is not met.

The function continues to the for loop and initializes val to one.

The line assigns one to the third element of the array. The reason this is the third element is because I N D is two. Remember, array index always starts at zero.

The next line is a function call. The first argument is the address of the heap memory. The second argument becomes three. The third argument is left minus val and it is one.

The function checks the stop condition. The third argument is one, not zero; thus, the stop condition is not met.

The function moves to the for loop and initializes val to one.

Inside the for loop, the fourth element of the array, of index three, is assigned to one.

The next line is a function call and a new frame is pushed to the top of the stack memory.

At this moment, the third argument is zero and the stop condition is met.

The function prints the values in the array. This output is one plus one plus one plus one.

The next line is return. This function call is ended.

The top frame is popped and val increases from one to two.

The condition of the for loop is that val must smaller than or equal to left. Left is one and val is two. Thus, the function exits the for loop.

There is nothing else to do after the for loop. This function call is ready to end.

The top frame is popped.

Val increases to two. In this frame, left is two. Thus, the program enters the for loop and assigns two to the third array element, with index equals to two.

The next line is a function call. The second argument is three and the third argument is left minus val equal to zero. A new frame is pushed to the top of the stack memory.

Now, the stop condition is met.

The program prints the first three elements in the array as one plus one plus two.

Please notice that the program prints only three numbers, not four.

The next statement is return. That means the function call ends.

The top frame is popped. Val increases to three.

Since val is greater than left now, the function exits the for loop.

The top frame is popped.

Val increases to two.

Left is three. Thus, the program continues in the for loop.

The program assigns two to the second array element within index equal to one.

The next line is a function call.

The first argument is the address of the array. The second argument is I N D plus one equal to two. The third argument is left minus val equal to one.

The stop condition is not satisfied.

The program enters the for loop and assigns one to val. .

This line assigns one to the array element with index equal to two.

The next line is a function call. The third argument is left minus val equals to zero.

The stop condition is met.

Thus, the program prints the first three elements one plus two plus one.

The next line is return. This function call ends.

The top frame is popped. Val increases to two.

Since val is greater than left, the program exists the for loop.

After the for loop, the function has nothing else to do. This function call ends.

The top frame is popped. Val increases to three.

Left is also three. Thus, the program enters the for loop.

The next line assigns three to the array element with index equals to one.

The next line is a function call. A frame is pushed.

The stop condition is met.

The program prints the first two elements of the array as one plus three.

The next line is return. The function call ends.

The top frame is popped. Val increases to four. Since it is greater than three, the program does not enter the for loop.

The function call ends.

The top frame is popped.

Val increases to two.

Left is four and is greater than val. Thus, the program enters the for loop.

The first line inside the for loop assigns two to the first element of the array.

This lecture explains step-by-step how a recursive function works. We closely examine the changes in the stack and the heap memory.

It is important for you to become very familiar with the changes in stack and heap memory if you want to fully understand recursive functions.