Fundamentals of Programming 2

Recursion, Divide-And-Conquer

Arkadiusz Chrobot

Department of Information Systems

March 11, 2024

Outline

Recursion

Introduction

In the previous semester we have discussed how functions in programs are performed. A crucial part in this operation has a stack, or more precisely, a call stack. It is an area in the program memory where data needed for performing a function are stored. The data are organized in *stack frames* also called *activation records*. Activation records are added to the call stack according to the LIFO principle. The stack frame stores a return address, local variables, parameters, among other data. Using a stack for handling function calls has interesting consequences. One of them is the possibility of creating *recursive subroutines*, that in case of the C language are *recursive functions*. A recursive function is a function that calls itself. Such functions implement *recursive algorithms*. These algorithms split the problem that they solve into finite number of subproblems of smaller size until they reduce them to a cases that can be directly solved. Then the algorithms combine the results to get the solution of the initial problem.

Recursion

Introduction

Let's analyse how the recursion works and discover its associations with the call stack, using as an example a recursive function that calculates the factorial. The definition of the factorial of a natural number n is as follows:

$$n! = \begin{cases} 1 \quad \text{if } n = 0 \text{ or } n = 1 \\ (n-1)! \cdot n \quad \text{for } n > 1 \end{cases}$$

It can be observed that the definition is recursive — for $n > 1$ the factorial can be calculated if the result of factorial for an argument less by 1 is already known. The definition also has two base cases, i.e. for which the result is given directly. These cases are for $n = 0$ and $n = 1$. A definition of function which implementation is based on the presented definition of factorial is in the next slide.

Recursion

Introduction — Factorial

```c
unsigned long int factorial(unsigned char n)
{
    if(n==0||n==1)
        return 1;
    else
        return factorial(n-1)*n;
}
```

Recursion

Introduction — Factorial

The function correctly calculates the factorial for a value of the parameter $n < 65$. Above that value the overflow of function result, which is of the `unsigned long int` type, occurs. Let's analyse how the function is performed for `n=3`. When it is invoked a stack frame is created for this function instance. The function checks the condition, which is false, so it tries to evaluate the expression `factorial(2)*3`. To complete the task it calls itself but for `n=2`. Just like in the previous call a stack frame is created for the new instance and the function check the condition in the line no. 3, which also is false. Then it tries to evaluate the `factorial(1)*2` expression. To this end it calls itself for `n=1`. Like previously a stack frame is created for this function call, but this time the condition in the third line is true and the instance of the function exits and returns the value of `1`.

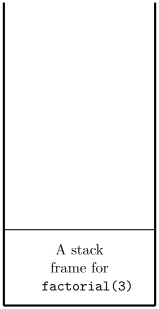
Recursion

Introduction — Factorial

After the function exits and returns the value of factorial for `n=1`, the control flow goes back to the instance of the function for `n=2`. This time the function can evaluate the `factorial(1)*2` expression, by replacing the first term of the expression with the value of `1`. After the result (`2`) is calculated the instance of the function returns it and exits. The control flow goes back to the instance of the function for `n=3`. This time it also can evaluate the `factorial(2)*3` expression by replacing the first term with the value of `2`. Finally, the function exits and returns `6` (the value of factorial for `n=3`). Any time each instance of the function exits, a stack frame is removed from the call stack. The content of the stack is depicted schematically in the next slide.

Recursion

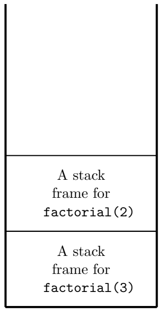
Introduction — Factorial



The function call for `n=3`

Recursion

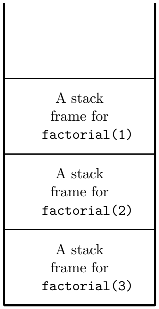
Introduction — Factorial


A stack frame for `factorial(2)`
A stack frame for `factorial(3)`

A recursive call: `factorial(2)*3`

Recursion

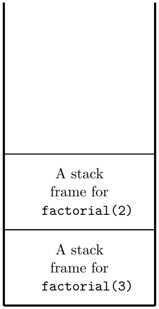
Introduction — Factorial


A stack frame for factorial(1)
A stack frame for factorial(2)
A stack frame for factorial(3)

A recursive call: `factorial(1)*2*3`

Recursion

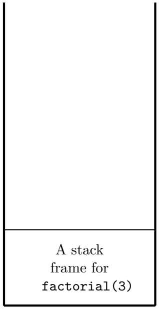
Introduction — Factorial


A stack frame for `factorial(2)`
A stack frame for `factorial(3)`

The `factorial(1)` call returns result an exits.

Recursion

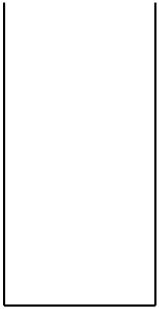
Introduction — Factorial



After `factorial(2)` exits.

Recursion

Introduction — Factorial



After `factorial(3)` exits.

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

```
factorial(3)
```

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{factorial(2)*3}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{factorial(2)*3} \arrow[d] \\
\texttt{factorial(1)*2}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{factorial(2)*3} \arrow[d] \\
\texttt{factorial(1)*2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{factorial(2)*3} \arrow[d] \\
\texttt{1*2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{factorial(2)*3} \arrow[d] \\
\texttt{2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{2*3} \arrow[d] \\
\texttt{2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)} \arrow[d] \\
\texttt{6} \arrow[d] \\
\texttt{2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

Recursion

Introduction — Factorial

Usually the performance of recursive functions is illustrated not by depicting the state of call stack, but by drawing a *function call tree* which is also, in this case, a *recursion tree*. In case of the `factorial()` function for `n=3` the recursion tree is very simple (the mathematicians would say that it is a degenerate tree).

$$\begin{tikzcd}
\texttt{factorial(3)=6} \arrow[d] \\
\texttt{6} \arrow[d] \\
\texttt{2} \arrow[d] \\
\texttt{1}
\end{tikzcd}$$

The Divide-And-Conquer Algorithm

The Divide-And-Conquer is a method of designing recursive algorithms. It consists of three steps:

Divide-And-Conquer

1. **Divide**
   Divide the problem into subproblems of the same type, but of a smaller size.
2. **Conquer**
   Solve the subproblems recursively, unless their size is so small, that they can be tackled with using direct methods.
3. **Merge**
   Combine the solutions of subproblems to get the solution of the original problem.

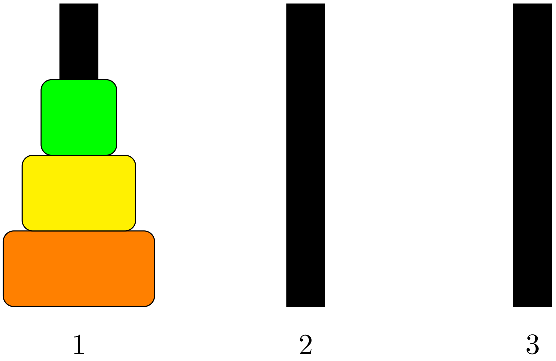
Divide-And-Conquer

Tower of Hanoi

The Divide-And-Conquer method can be applied to solve the Tower of Hanoi problem, which has been formulated in the year 1883 by a French mathematician Édouard Lucas. In the problem a tower is given that consists of discs of different diameters located on a single peg, and so constructed that smaller discs lie on bigger discs. Also a two additional empty pegs are given. The goal of the problem is to move the tower to one of the two additional pegs, using the remaining one in the process, if necessary, but during each step only one disk can be moved and never a bigger disk can be put on a top of a smaller disk. An animation in the next slide shows the solution of the problem for a tower consisting of three discs.

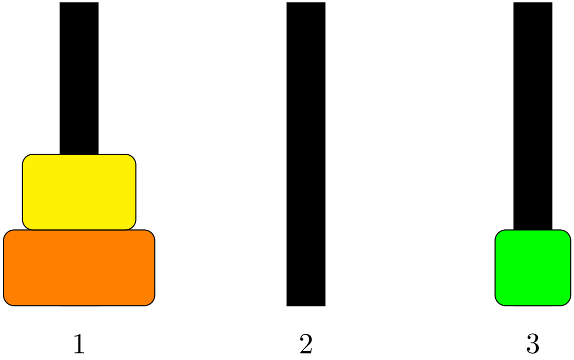
Divide-And-Conquer

Tower of Hanoi



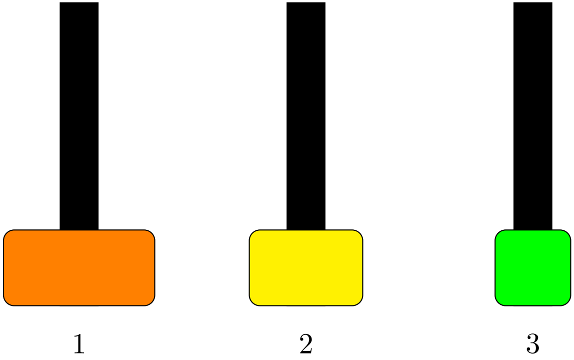
Divide-And-Conquer

Tower of Hanoi



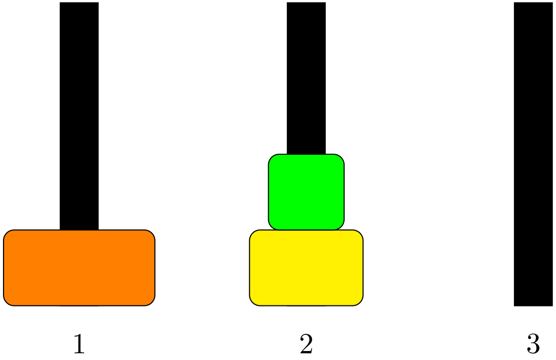
Divide-And-Conquer

Tower of Hanoi



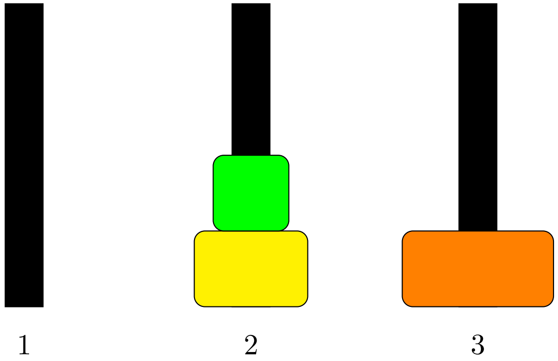
Divide-And-Conquer

Tower of Hanoi

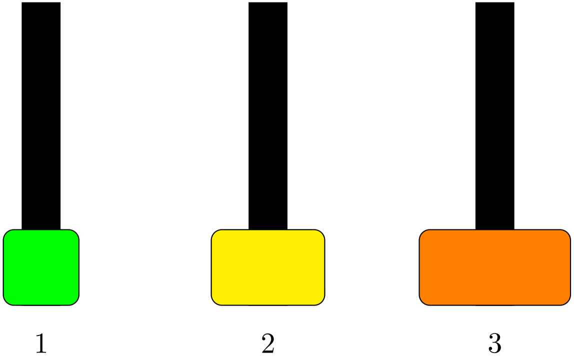


Divide-And-Conquer

Tower of Hanoi

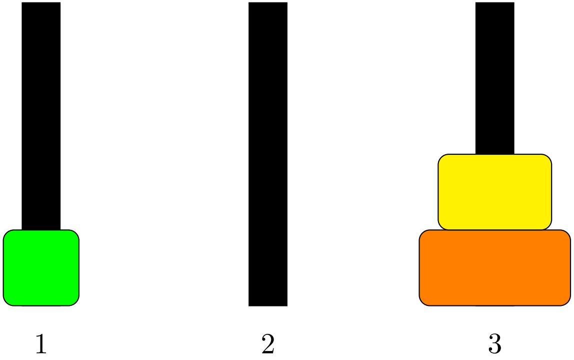


Divide-And-Conquer
Tower of Hanoi



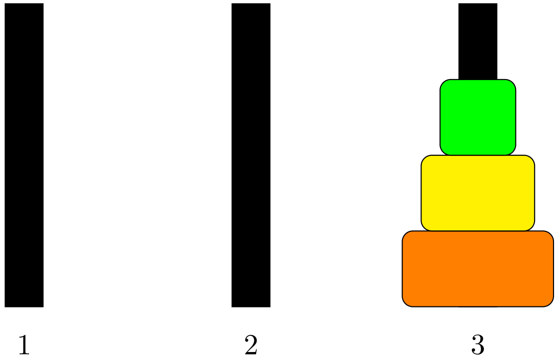
Divide-And-Conquer

Tower of Hanoi



Divide-And-Conquer

Tower of Hanoi



Divide-And-Conquer
Tower of Hanoi

Actually, a two problems of Tower of Hanoi will be solved in the lecture. The first one has been formulated in the book “Concrete Mathematics” by D.E.Knuth, R.L.Graham and O.Patashnik (Addison-Wesley Publishing, 1989):

The First Problem of Tower of Hanoi

Knowing how many discs the tower has, calculate the minimal number of steps required to move it to the destination peg.

Divide-And-Conquer

Tower of Hanoi

The problem can be solved by applying the Divide-And-Conquer method. According to its description, in the first step a partition of the problem into subproblem of smaller size has to be found. In the case of Tower of Hanoi the task is pretty simple — the tower consists of discrete discs, so the subproblem of directly smaller size, than the problem of moving n discs, is to move $n-1$ discs. In the second step the strategy of recursively solving the subproblems has to be defined. Moving the n discs is straightforward, assuming that the method for moving $n-1$ disks is known:

1. Move $n-1$ discs from the source peg to the helper peg.
2. Move the nth disc from the source peg to the destination peg.
3. Move $n-1$ discs from the helper peg to the destination peg.

Divide-And-Conquer

Tower of Hanoi

Let's denote the minimal number of steps required to move the Tower of Hanoi of n disks by T_n, and by T_{n-1} the minimal number of necessary steps to move the Tower of Hanoi of $n-1$ disks. From the previous slide it can be derived that $T_n = 2 \cdot T_{n-1} + 1$. This relation for towers of n and $n-1$ disks is also true for towers of $n-1$ and $n-2$ disks, i.e. knowing how many steps it takes to move the tower of $n-2$ disks, it is possible to calculate how many steps it takes to move the tower of $n-1$ disks. Now, the base case has to be defined. It can be assumed that it is a problem of moving an empty tower, i.e. which has $n = 0$ disks. In that case no steps are required to move the tower, which can be denoted as $T_0 = 0$. The Merge Step is quite simple in this problem: having a solution for `0` disks the solution for `1` disk can be found, then for `2` disks and so on. The next slide contains a definition of a function that implements this algorithm.

Divide-And-Conquer

Tower of Hanoi

```
unsigned long int find_hanoi_steps(unsigned char discs)
{
    if(discs==0)
        return 0;
    else
        return 2*find_hanoi_steps(discs-1)+1;
}
```

Divide-And-Conquer

Tower of Hanoi

The number of discs of Tower of Hanoi is passed through the `discs` parameter to the function. Please observe, that using the Divide-And-Conquer method makes it possible to write a function that calculates the factorial. It means that the method may be applied to a variety of problems. The next slide shows the recursion tree for the `find_hanoi_steps()` function invoked for a Tower of Hanoi with 4 disks.

Divide-And-Conquer

Tower of Hanoi

```
find_hanoi_steps(4)
```

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}[row sep=small]
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(1)+1}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(1)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(0)+1}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(1)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(0)+1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(1)+1} \arrow[d] \\
\texttt{2*0+1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(1)+1} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}[row sep=small]
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{2*1+1} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(2)+1} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}[row sep=large]
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{2*3+1} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}[row sep=large]
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*find\_hanoi\_steps(3)+1} \arrow[d] \\
\texttt{7} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{2*7+1} \arrow[d] \\
\texttt{7} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}[row sep=large]
\texttt{find\_hanoi\_steps(4)} \arrow[d] \\
\texttt{15} \arrow[d] \\
\texttt{7} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

$$\begin{tikzcd}
\texttt{find\_hanoi\_steps(4)=15} \arrow[d] \\
\texttt{15} \arrow[d] \\
\texttt{7} \arrow[d] \\
\texttt{3} \arrow[d] \\
\texttt{1} \arrow[d] \\
\texttt{0}
\end{tikzcd}$$

Divide-And-Conquer

Tower of Hanoi

Let's now try to find solution for the second problem of Tower of Hanoi which is defined as follows:

Second Problem of Tower of Hanoi

Find an algorithm for moving the Tower of Hanoi with minimal number of steps.

It turns out, that the same way of reasoning that has been applied for solving the first problem of Tower of Hanoi can also be used for finding the solution for the second one. Like in the previous problem, moving an empty tower requires no action, so `0` steps has to be made. To move a tower of n discs, first a tower of $n-1$ disc has to be moved from the source peg to the helper peg. Next, the `nth` disc has to be moved form the source peg to the destination peg, and finally the tower of $n-1$ discs has to be moved from the helper peg to the destination peg. The next slide contains a definition of a function that implements this algorithm.

Divide-And-Conquer

Tower of Hanoi

```c
void hanoi_movements(unsigned int discs,
                     unsigned char source,
                     unsigned char helper,
                     unsigned char destination)
{
    if(discs) {
        hanoi_movements(discs-1,source,destination,helper);
        printf("Move the disc no. %u from the peg no. %u\
        to the peg no. %u\n",discs,source,destination);
        hanoi_movements(discs-1,helper,source,destination);
    }
}
```

Divide-And-Conquer

Tower of Hanoi

By the first parameter the number of Tower of Hanoi discs is passed to the function. By the next three parameters the number of the source peg, the helper peg and the destination peg are passed. In the line no. 6 the function checks if the number of discs is greater than zero. If the condition is satisfied then it starts informing user which disc (by displaying its number) has to be moved from which peg to which peg (also by displaying their numbers) in a given step. According to the presented analysis, first the function has to move the smaller tower from the source peg to the helper peg, which is in this case a destination peg. Hence, the first recursive invocation of the function (line no. 7). Next, the function displays a message informing the user to move the n`th` disc (lines no. 8 and 9) and finally it calls itself recursively (line no 10) to move $n - 1$ discs from the helper peg (which in this case is the source peg) to the destination peg.

Pros and Cons of Recursion

The biggest advantage of using the recursion is that it makes the definitions of functions brief. Let's check it using two examples. The first example is a function that converts decimal numbers greater than `0` into binary numbers. The second example is the Quicksort algorithm and its implementation in a form of two functions.

Pros and Cons of Recursion

Conversion from Decimal to Binary

At the first sight the algorithm of converting a decimal number into a binary number seems not to be recursive. It consists of finding the reminder after division of a number by two and the integer result of dividing the number by two, until the latter operation yields `0`. Then the reminders has to be ordered from the last to the first. The last sentence suggests using a stack, which means that applying a recursion would simplify the solution — the recursive functions use the call stack. Please observe, that the result of integer division becomes the input data for the next step of the algorithm, so it may be expressed in a recursive form. The base case is when the integer division yields zero. The next slide presents a definition of a function which converts a decimal number into binary and displays the result on the screen.

Pros and Cons of Recursion

Conversion from Decimal to Binary

```c
void convert_to_binary(unsigned long int number)
{
    if(number) {
        convert_to_binary(number/2);
        printf("%lu",number%2);
    }
}
```

Pros and Cons of Recursion

Conversion from Decimal to Binary

The presented function check if the number passed to it is greater then zero. If so, it calls itself recursively for this number divided by two. This sequence of calls ends when the value of the number becomes zero for some invocation. Then returns from the recursive calls take place and in each of them the `printf()` function is invoked which displays the reminder after division of the number passed by the parameter by two. The algorithm that the function implements may be created with the use of Divide-And-Conquer method. The base case is when the integer division yields zero. The partition of the problem is made each time the number is divided by two. The merging takes place when the function prints the reminders. The function doesn't convert the `0` number. It could be changed by placing only the recursive invocation in the conditional statement, but after such a modification the function will always display the binary numbers that start with a leading zero (as a value of the most significant bit).

Pros and Cons of Recursion

Quicksort

The Quicksort algorithm was developed by a British computer scientist C.A.R. Hoare and it is one of the most efficient sorting algorithms. Although its worst-case time complexity is $\Theta(n^2)$, where n is the number of element, its average and best-case time complexity is $\Theta(n \cdot log_2(n))$. Constants hidden by the asymptotic notation are small. The Quicksort is an in-place sorting algorithm, but its space complexity is $O(n)$. It is a consequence of the fact, that it is a recursive algorithm implemented in a form of a recursive subroutine, hence it intensively uses the call stack. It can be implemented as an iterative subroutine, but this proves to be a challenging task and the iterative implementation isn't more effective than the recursive one. The Quicksort performs unstable sorting.

Quicksort and "Divide and Conquer"

The Quicksort algorithm can be described using the "Divide and Conquer" method:

Divide: The $A[p \dots r]$ array is partitioned (values of its elements are swapped) into two nonempty parts $A[p \dots q]$ and $A[q+1 \dots r]$, such that a value of each element in $A[p \dots q]$ is not greater than the value of any element in $A[q+1 \dots r]$. The q index is determined by a partitioning subroutine.

Conquer: The two parts: $A[p \dots q]$ and $A[q+1 \dots r]$ are sorted by applying the Quicksort algorithm recursively.

Merge: Since the Quicksort is an in-place algorithm, no additional steps are required to merge the sorted parts: the whole $A[p \dots r]$ array is already sorted.

Pros and Cons of Recursion

Quicksort — The `quicksort()` Function

```
void quicksort(int_array_type array, int low, int high)
{
    if(low<high) {
        int partition_index = partition(array,low,high);
        quicksort(array, low, partition_index);
        quicksort(array, partition_index+1, high);
    }
}
```

Pros and Cons of Recursion

Quicksort — The `quicksort()` Function

The `quicksort()` function corresponds to the **Conquer** step in the description presented in the previous slide. It doesn't return any value, but has three parameters. The first one is used for passing the array. By the second and third parameters are passed the indices that specify the area of the array that has to be sorted. Initially, when the `quicksort()` function is called, for example, in the `main()` function, this area covers the whole array. Inside the body of the `quicksort()` function the first index (`low`) is compared with the last index (`high`). If the former is less than the latter then there is a part (area) of the array that still needs to be sorted. Otherwise the function exits. If the condition in the 3rd line is satisfied then the `partition()` function is invoked that reorders the given part of the array and determines the point where this area is partitioned in two smaller parts. Next, the `quicksort()` function is called twice, for each of the new parts separately.

Pros and Cons of Recursion

Quicksort — The `quicksort()` Function

The first part is sorted by an instance of the `quicksort()` function that deals with elements of the array that have indices raging from the first index (`low`) to the partition index (`partition_index`), including both of them. The second part consists of elements with indices raging from the partition index (excluding) to the last index (`high`), including. The `partition()` function is defined in the program before the `quicksort()` function, but in the slides it is described after the latter.

Pros and Cons of Recursion

Quicksort — The partition() Function

```c
int partition(int_array_type array, int low, int high)
{
    int pivot = array[low];
    int i = low-1, j = high+1;

    while(i<j) {
        while(array[--j]>pivot)
            ;
        while(array[++i]<pivot)
            ;
        if(i<j)
            swap(&array[i],&array[j]);
    }
    return j;
}
```

Pros and Cons of Recursion

Quicksort — The `partition()` Function

The `partition()` function corresponds to the **Divide** step in the description that uses the "Divide and Conquer" method. It has three parameters, which have the same meaning as the parameters of the `quicksort()` function. The `partition()` function returns a number, which is an index that specifies the partition point of the currently sorted part of the array. In the 3rd line of the function is declared and initialised a variable named `pivot`, that stores so-called *pivot value* which specifies how the part of the array is reordered. In the function, the value of the first element of the sorted part of the array is assumed as the pivot value (line no. 3). In the 4th line of the function are declared and initialised two variables that are used for indexing the sorted part of the array from the beginning (the `i` variable) and from the end (the `j` variable). Please note, that initially both indices specify elements that are *outside* the sorted part of the array.

Pros and Cons of Recursion

Quicksort — Funkcja `partition()`

The outer `while` loop (6th line) is repeated as long as the the value of the `i` index is smaller than the value of the `j` index, or in other words, until the indices “meet” or “miss” each other. Inside the loop are performed two other `while` loops. The first one (lines no. 7 and 8) traverses the given part of an array starting from the end toward the beginning and searches for an element that has a value equal to or smaller than the pivot value. The second internal loop (lines no. 9 and 10) traverses the same part of the array but from the beginning to the end and searches for an element with a value greater than or equal to the pivot value. Please note, how these loops are implemented. The searching takes place inside the condition statement of the loops. The pre-increment and pre-decrement operators are applied to the indices to avoid accessing elements of the array that are outside of the sorted part or even accessing elements outside the array itself.

Pros and Cons of Recursion

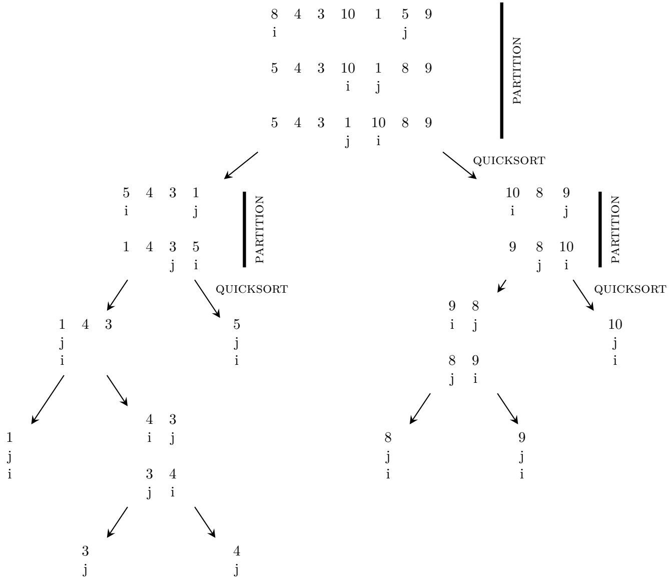
Quicksort — The `partition()` Function

After both internal loops stop, the function performs the conditional statement (11th line) to check whether the `i` index is smaller than the `j` index. If so, then the order of the values in the elements associated with these indices is incorrect, and they have to switch the places. If not, then the outer loop stops and the `j` index specifies a partition point for the sorted part of the array, hence the index is returned by the function (14th line).

In the next slide is a call tree that illustrates how the `quicksort()` function sorts an array that has seven elements that store natural numbers. In the upper part of the tree, it is marked which actions are performed by the `quicksort()` function and which by the `partition()` function. In the bottom part of the tree, no such description is given, to keep the drawing more legible.

Pros and Cons of Recursion

Quicksort — Call Tree



Pros and Cons of Recursion

Effectiveness of the Recursion

Let's consider the effectiveness of the recursion. Each recursive call involves creating a stack frame. It takes time and uses some free space in the memory, so it degrades the effectiveness of recursive functions. The more recursive calls a function has the less efficient it becomes, comparing to its iterative equivalent. There is one more reason for refraining from using the recursion in every possible function. It is explained with the use of Fibonacci Numbers problem.

Pros and Cons of Recursion

Fibonacci Sequence

The Fibonacci Sequence has been discovered by a medieval Italian mathematician Leonardo Fibonacci, who tried to use it for describing the grow of a population of rabbits. It occurred to be insufficient for this purpose, but it can be applied to many other practical problems. Even some concepts from art can be described using this sequence. The Fibonacci Sequence is defined as follows:

$$fibonacci(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fibonacci(n-2) + fibonacci(n-1) & \text{for } n \geq 2 \end{cases}$$

The n in the formula is a natural number and it defines the place of a given Fibonacci Number in the sequence. All numbers that belong to the Fibonacci Sequence are natural numbers. The definition of the sequence encourages to implement a function that finds successive Fibonacci Numbers in a recursive fashion. A definition of such a function in presented in the next slide.

Pros and Cons of Recursion

Fibonacci Sequence

```c
unsigned int get_fibonacci_number(unsigned char order)
{
    if(order==0)
        return 0;
    if(order==1)
        return 1;
    return get_fibonacci_number(order-1)+get_fibonacci_number(order-2);
}
```

Pros and Cons of Recursion

Fibonacci Sequence

The definition of the function is brief and understandable because it mirrors the definition of the sequence. The analysis of the function behaviour is a little more difficult since it calls itself twice in the 7th line. In that case it could be assumed that expressions are evaluated from the left to the right side, so the “left” invocation is performed as first, and only after it exits the “right” is performed. After the latter exits the total value of the expression can be calculated. Let’s create a recursion tree for the function when it is invoked with the `order` parameter equal `4`. For the sake of simplicity the name of the function in the tree is shortened to just one letter — `f`.

Pros and Cons of Recursion

Fibonacci Sequence

```
f(4)
```

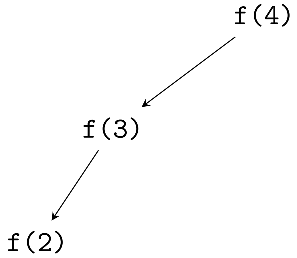
Pros and Cons of Recursion

Fibonacci Sequence

$$\begin{tikzcd}
 & \texttt{f(4)} \arrow[dl] \\
\texttt{f(3)} &
\end{tikzcd}$$

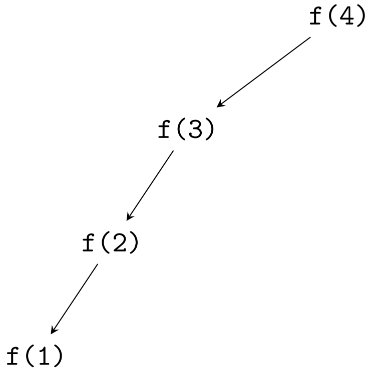
Pros and Cons of Recursion

Fibonacci Sequence


$$\begin{tikzcd}
 & & \texttt{f(4)} \arrow[dl] \\
 & \texttt{f(3)} \arrow[dl] & \\
\texttt{f(2)} & &
\end{tikzcd}$$

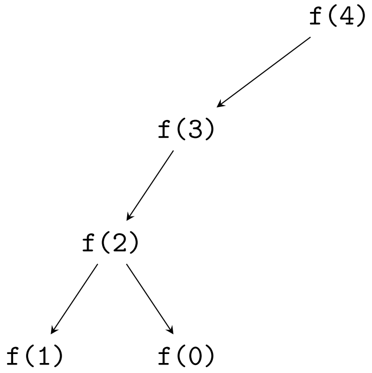
Pros and Cons of Recursion

Fibonacci Sequence



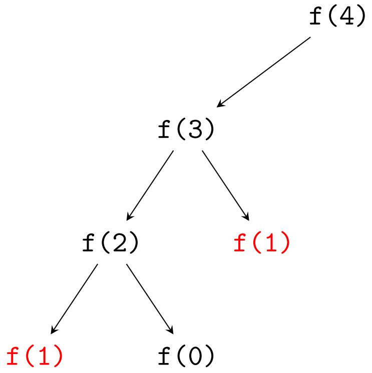
Pros and Cons of Recursion

Fibonacci Sequence



Pros and Cons of Recursion

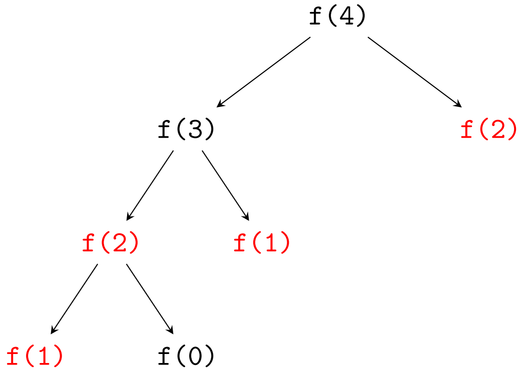
Fibonacci Sequence



$$\begin{tikzcd}
 & & & f(4) \arrow[dl] \\
 & & f(3) \arrow[dl] \arrow[dr] & \\
 & f(2) \arrow[dl] \arrow[dr] & & f(1) \\
f(1) & & f(0) &
\end{tikzcd}$$

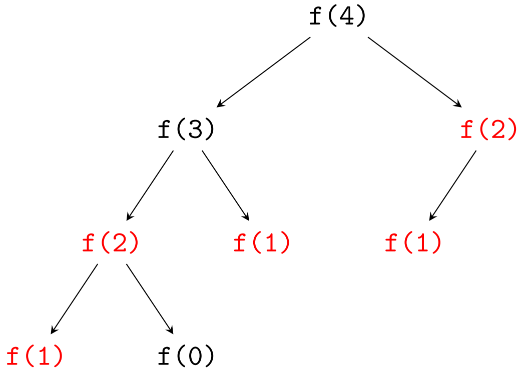
Pros and Cons of Recursion

Fibonacci Sequence



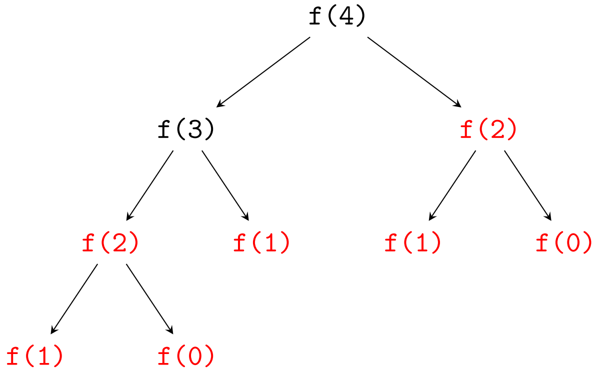
Pros and Cons of Recursion

Fibonacci Sequence



Pros and Cons of Recursion

Fibonacci Sequence


$$\begin{tikzcd}
 & & & f(4) \arrow[dll] \arrow[drr] & & & \\
 & f(3) \arrow[dl] \arrow[dr] & & & & f(2) \arrow[dl] \arrow[dr] & \\
f(2) \arrow[d] \arrow[dr] & & f(1) & & f(1) & & f(0) \\
f(1) & f(0) & & & & &
\end{tikzcd}$$

Pros and Cons of Recursion

Fibonacci Sequence

It can be observed, by analysing the recursion tree, that a lot of the recursive calls calculate the same value (they are marked in red color in the tree). This influences the effectiveness measured as a time of performance and the required memory. Finding the first 45 Fibonacci Numbers with the use of the function takes a few second, even with the use of a powerful computer.
The question arises, if a more efficient version of the function can be created, that doesn't perform the redundant calculations. The answer is positive. According to the Fibonacci Sequence definition, the function needs only to remember the two last Fibonacci Numbers to calculate the next one. Moreover it doesn't have to be recursive, it can be an iterative function. The definition of such a function is presented in the next slide.

Pros and Cons of Recursion

Fibonacci Sequence — Iterative Version

```
unsigned int get_fibonacci_number(unsigned char order)
{
    unsigned int current = 0, next = 1, result = 0;
    unsigned int i;
    for(i=0;i<order;i++) {
        result = current + next;
        current = next;
        next = result;
    }
    return current;
}
```

Pros and Cons of Recursion

Fibonacci Sequence

The definition of the function is not as legible as the definition of its recursive version, but still pretty understandable. The `current` variable stores the result of the currently calculated Fibonacci Number and the `next` variable the value of the next Fibonacci Number. To calculate another Fibonacci Number these two values are added. The result is stored in the `result` variable. Then the value stored in `next` variable replaces the one stored in the `current` variable, and the value stored in the `result` variable replaces the one stored in the `next` variable. These calculations are preformed in the `for` loop which repeats them as many times as it is defined by the position in the sequence of the number to be calculated. The loop also calculates a redundant Fibonacci Number, but only one. It's the number that follows in the sequence the one that is calculated. Calculating the initial 45 Fibonacci Numbers with this function, using the same computer as previously takes a lot less time. Additionally, the function requires less memory while running.

Pros and Cons of Recursion

Fibonacci Sequence

It can be proved, using the computational complexity theory, that the algorithm applied by the recursive function, which calculates the Fibonacci Number, is an exponential-time algorithm. For the forth Fibonacci Number it creates a call tree with four levels. For the third Fibonacci Number it creates a call tree of three levels. Because each node of the tree has at most two successors then the total number of nodes in the tree is 2^{order}, where the `order` is the position in the Fibonacci Sequence of the calculated Fibonacci Number. Additionally, the bigger the call tree, the more it contains instances[1] of the function that calculate redundant values. The iterative version utilises a linear-time algorithm, thus its running time is much shorter, and it uses less memory than the recursive version.

[1] A function instance is an invocation of the function with a specific argument.

Pros and Cons of Recursion

Binomial Coefficients

Calculating the binomial coefficients, which are commonly used in combinatorics, requires applying a complicated formula. However, the operation has some properties that makes it possible to calculate binomial coefficients recursively:

$$\binom{n}{k} = \begin{cases} 1 \quad \text{for } k = 0 \;\; \texttt{or} \;\; k = n \\ n \quad \text{for } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{in other cases} \end{cases}$$

The next slides contains a function definition that implements this formula, similarly like the recursive function that calculated Fibonacci Numbers applied the Fibonacci Sequence definition.

Pros and Cons of Recursion

Binomial Coefficients

```c
unsigned int get_binomial_coefficients(unsigned char n, unsigned char k)
{
    if(k==0||k==n)
        return 1;
    if(k==1)
        return n;
    return get_binomial_coefficients(n-1,k-1)+get_binomial_coefficients(n-1,k);
}
```

Pros and Cons of Recursion

Binomial Coefficients

Using the recursion is not the most effective way of calculating the binomial coefficients, just like applying it for calculating Fibonacci Numbers. Aside from recursive calls that calculate redundant values, in the case of the function presented in the previous slide, a risk of overflow of the `unsigned int` type values arises. This error leads to incorrect results. However, creating an iterative equivalent for the function is not as easy as in the case of Fibonacci Numbers. The iterative function requires applying of two-dimensional array and two loops, nested one in the other.

Common Mistakes

Using the recursion is prone to many mistakes. The most common category of such mistakes are incorrectly defined conditions for terminating the recursion. Such mistakes can be of mathematical or computing science nature. In the first case the base cases and the problem partitioning operation are incorrectly defined and as a result the condition for stopping the recursion is never met. In the second case the data types of variables may be chosen incorrectly or the statements used in the function body may not behave as intended by the programmer. Wrongly chosen data types can cause overflows and thus the function may never terminate. Similarly, using wrong statements, may cause the same troubles. Especially "treacherous" are the increment and decrement operators. Using them in arguments for recursive calls should be avoided. Sometimes the cause of an error may be misinterpretation of the function code by the compiler. That's why paying an attention to warnings issued by this program during the compilation process is important.

Common Mistakes

The recursion has a lot in common with the mathematical induction. The correctness of recursive algorithms may be proven with the use of this technique. The mistakes of computing science nature can be dealt with by using such tools as debuggers and by paying attention to details while writing recursive functions. In the ideal world errors in recursive functions would result in infinite recursive calls of these functions. In the real world such incorrectly written recursive functions result in a *stack overflow*, which means that the function has generated so many stack frames that there is no space left on the stack for a new one. This exception can be also caused by correctly written recursive functions, but run in environments in which they have only a stack of a small size to their disposal. In that case replacing the arguments passed by a value by the arguments passed by a pointer or by a constant is worth considering.

Summary

The Divide-And-Conquer method is a powerful tool for creating recursive algorithms. The recursion makes it possible to implement them in a compact form. However, as it is demonstrated in the lecture, using recursive functions is not always efficient. Sometimes it is better to spend some time analysing the problem in order to find an iterative or even simpler implementation of the solution. For example by reading the book “Concrete Mathematics”, a simple solution of the first problem of Tower of Hanoi can be found. The recursive equations describing the number of minimal steps required to move the tower can be simplified to $T_n = 2^n - 1$, where n is the number of discs in the tower. That means that even a loop is not necessary to calculate the result. A function that applies such a formula is sufficient. Moreover, it is a function that utilises a constant-time algorithm.

Summary

Although recursive functions have drawbacks, learning this technique is important to any decent programmer. There are recursive algorithms which cannot be easily implemented in an iterative form. Such an implementation would require to explicitly implement a stack as, for example, a dynamically allocated data structure, which is a more tedious work than just simply writing a recursive function. Some of the recursive functions can be automatically converted to an iterative form by the compiler. Finally, the last argument for using the recursion is that there are some efficient recursive algorithms that are easily implemented in a form of recursive functions. The iterative version of these functions are more complicated and at most as efficient as the recursive ones. An example of such an algorithm is *QuickSort* which has been described in this lecture.

Questions

?

The End

Thank You For Your Attention!