Programmering for computerteknologi Hand-in Assignment Exercises

Week 9: Functions that call themselves

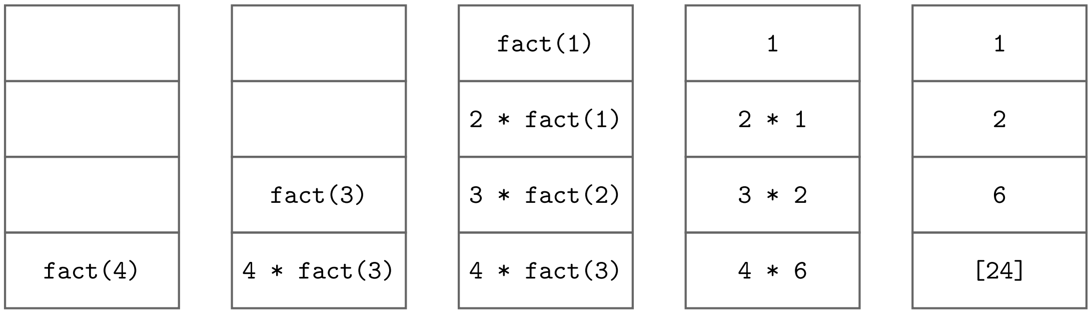
Written by: Alexander A. Christensen (202205452)

Disclaimer: Due to errors with CMake that neither me, nor the TAs have solved, the test-cases have not been run. Instead, the functions have been manually tested.

The code can still be found at https://github.com/Aarhus-University-ECE/assignment-9-A-CHRI

Exercise 1

When calling `fact(4)`, we can visualize the process, as the function calls itself to return the answer.



Looking at the stacks from left to right, we can see the various calculation done to compute `fact(4)`.

Exercise 2

We wish to implement a script which sums an array using recursion. The following recursive equation describes the function.

$$\text{sum}(a,n) = \begin{cases} a[n-1] + \text{sum}(a, n-1), & \text{if } n > 0 \\ 0, & \text{if } n = 0 \end{cases}$$

This can be implemented as shown below.

```c
int sum(int a[], int n)
{
    if (n == 0)
    {
        return 0;
    }
    else
    {
        return a[n - 1] + sum(a, n - 1);
    }
}
```

Exercise 3

We wish to implement a recursive search, that does not require the list of elements to be sorted. The recursive equation used is as follows

$$\text{search}(a, n, x) = \begin{cases} \text{true}, & \text{if } n > 0 \text{ and } a[n-1] == x \\ \text{search}(a, n-1, x), & \text{if } n > 0 \text{ and } a[n-1]! = x \\ \text{false}, & \text{if } n = 0 \end{cases}$$

The function has been implemented below.

```c
bool search(int a[], int n, int x)
{
    if (n == 0)
        return false;
    else
    {
        if (a[n - 1] == x)
            return true;
        search(a, n - 1, x);
    }
}
```

Exercise 4

DFS can be implemented using a stack, using the following princibles.

1. Pop the top of the stack and visit it
2. Push the previous nodes children to the stack
3. Repeat as long as there is items in the stack

This can be implemented using the following function.

```c
void DFT(node *root)
{
  /* Initiate the stack */
  stack *top = NULL;

  /* Push the root node to the stack */
  top = push(top, root);

  /* While the stack is not empty: */
  while (top != NULL)
  {
    /* Create pointers for left and right child */
    node *left = top->node->lchild;
    node *right = top->node->rchild;

    /* Pop and visit the top node in the stack */
    top->node->visited = true;
    top = pop(top);

    /* Add the children to the stack, if there is children */
    if (right != NULL)
      top = push(top, right);
    if (left != NULL)
      top = push(top, left);
  }
}
```

Here we've created a struct `stack`, which contains the top node, as well as a pointer to the next node in the stack. These can be linked together to create the stack.

Besides this we use the `pop` and `push` functions.

```c
    stack *push(stack *topp, node *node)
{
  /* Allocate memory for the new stack element */
  stack *new = (stack *)malloc(sizeof(stack));

  /* Add the stack element to top of the stack */
  new->next = topp;
  new->node = node;

  return new;
}
```

```c
stack *pop(stack *topp)
{
  /* Grab the top element*/
  stack *t = topp;
  printf("%d ", t->node->num);

  /* Detach the top element from the stack */
  topp = topp->next;
  free(t);

  /* Return the stack */
  return topp;
}
```