L14 - Recursion

December 18, 2019

0.1 Recursion

Administrivia

- wed Dec 4: class cancelled
- Fri Dec 6: no lecture; Monday schedule
- Final exam: comprehensive; weighted toward 2nd half of course

0.2 Basics of recursion

In order to write a recursive function, we need two different cases:

- the *base case*: a degenerate case where the answer is obvious
- the *recursive case*: a case where we can shrink the problem, towards the base case

If we do not reduce the problem size on each recursion or we cannot reach the base case, we will get *infinite recursion*, just like a `while()` loop that never reaches its end.

0.3 Calculating Factorial

How do we calculate 6!? We repeatedly calculate partial products:

$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

```
In [3]: // calculate n! iteratively
        #include <stdio.h>
        #include <stdlib.h>

        int factorial (int n)
        {
            int result = 1;

            for (int i = 1; i <= n; i++)
            {
                result = result * i;
            }

            return result;
        }
        int main()
        {
            int n = 6;
            printf("factorial: %d\n", factorial(n));
            return 0;
        }
```

```
factorial: 720
```

```
In [36]: // calculate n! recursively
         #include <stdio.h>
         #include <stdlib.h>

         long long int factorial(long long n)
         {
             if (n == 1)
             {
                 // the base case of factorial
                 printf("Calling factorial(1); returning 1\n");
                 return 1;
             }
             else
             {
                 // not the base case; call factorial again on a smaller problem
                 printf("Calling factorial(%lld)\n", n);
                 return n * factorial(n-1);
             }
         }

         int main()
         {
             long long n = 6;
             printf("factorial: %lld\n", factorial(n));
             return 0;
         }
```

```
Calling factorial(6)
Calling factorial(5)
Calling factorial(4)
Calling factorial(3)
Calling factorial(2)
Calling factorial(1); returning 1
factorial: 720
```

0.4 Calculating the sum of an array

We can calculate the sum of integers in an array recursively:

- base case: the sum of an array of one element is the value of that element
  - `sum(a[0] ... a[0]) = a[0]`
- recursive case: the sum of an array is the sum of `a[0]` and the sum of the rest of the array
  - `sum(a[0] ... a[n-1]) = sum(a[0] ... a[n-2] + a[n-1]`

```
In [41]: // calculate an array sum recursively
         #include <stdio.h>
         #include <stdlib.h>

         int sum(int arr[], int n)
         {
             if (n == 1) // base case; array length is 1
             {
                 return arr[0];
             }
             // recursive case; add last element to the sum of the rest of the array
             return arr[n-1] + sum(arr, n-1);
         }

         int main()
         {
             int a[] = {1, 2, 3, 4, 5};

             printf("array sum: %d", sum(a, 5));
         }
```

```
array sum: 15
```

0.5 Traversing a linked list recursively

Recall that we can traverse a linked list with the following for loop:

```
In [ ]: // includes and struct declaration here

        node_t* curr;

        while ( curr != NULL )
        {
            curr = curr->next;
            do_something();
        }
```

```c
// equivalent
for ( node_t* curr = head; curr != NULL; curr = curr->next)
{
    do_something();
}
```

Let's traverse the list with a recursive function instead:

In [50]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// node structure
struct node {
    int value;              // list payload
    struct node* next;      // pointer to the next node
};
typedef struct node node_t;

// node constructor function
node_t* node_construct( int value, node_t* next )
{
    node_t* p = malloc(sizeof(node_t));  // A: alocate memory
    assert ( p != NULL );
    p->value = value;                    // B: build structure (initialize values)
    p->next = next;                      // C: connect to next node
    return p;
}

void traverse( node_t* curr )
{
    if (curr == NULL)
    {
        return; // base case; go nowhere and do nothing
    }
    // recursive case
    printf("%d\n", curr->value);
    return traverse(curr->next);
}

int main()
{
    node_t* first;
    node_t* second;
    node_t* third;

    // initialize nodes
    third = node_construct(3, NULL);
    second = node_construct(2, third);
    first = node_construct(1, second);

    // get a pointer to the front of the list
    node_t* current = first;

    traverse(current);

    return EXIT_SUCCESS;
}
```

```
1
2
3
```

In [4]:

```
// an even shorter traverse() function
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// node structure
struct node {
    int value;              // list payload
    struct node* next;      // pointer to the next node
};
typedef struct node node_t;

// node constructor function
node_t* node_construct( int value, node_t* next )
{
    node_t* p = malloc(sizeof(node_t));  // A: alocate memory
    assert ( p != NULL );
    p->value = value;                    // B: build structure (initialize values)
    p->next = next;                      // C: connect to next node
    return p;
}

void traverse( node_t* curr )
{
    // even shorter
    if (curr != NULL)
    {
        printf("%d\n", curr->value);
        traverse(curr->next);
    }
}
```

```c
int main()
{
    node_t* first;
    node_t* second;
    node_t* third;

    // initialize nodes
    third = node_construct(3, NULL);
    second = node_construct(2, third);
    first = node_construct(1, second);

    // get a pointer to the front of the list
    node_t* current = first;

    traverse(current);

    return EXIT_SUCCESS;
}
```

```
1
2
3
```

0.6 Towers of Hanoi

From TutorialsPoint:

```
|           |          |
|           |          |
|           |          |
|          {=}         |   // smallest disk
|         {===}        |
|        {=====}       |   // largest disk
```

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are:

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

We can implement the Towers of Hanoi algorithm using *two* recursive cases:

- if there is exactly one disk, move it from the source to the destination
- if there are two disks:

- move the top disk to source to temp
- move the next disk from source to destination
- move the first disk from temp to destination

In [24]:
```c
#include <stdio.h>
#include <stdlib.h>

// disk: target disk
// source: tower where target disk is
// dest: tower to move the target disk to
// temp: the spare tower

void Hanoi(int disk, char source, char dest, char temp, int* count)
{
    if (disk == 1)
    {
        // move disk
        printf("Move disk 1 from tower %c to tower %c.\n", source, dest);
        *count = *count + 1;
        return;
    }

    /*
    neither of the above can be true to get here

    need to move at least two disks:
    - move smallest disk to the spare
    - move target disk to the destination
    - move smallest disk to the destination
    */
    Hanoi( disk - 1, source, temp, dest , count);
    printf("Move disk %d from tower %c to tower %c.\n", disk, source, dest);   // not
    Hanoi( disk - 1, temp, dest, source, count);
    *count = *count + 1;
}

int main()
{
    // count number of moves on heap
    int* moves = malloc(sizeof(int));
    *moves = 0;

    Hanoi( 4, 'A', 'B', 'C', moves);

    printf("Done in %d moves.\n", *moves);

    return EXIT_SUCCESS;
}
```

```
Move disk 1 from tower A to tower C.
Move disk 2 from tower A to tower B.
Move disk 1 from tower C to tower B.
Move disk 3 from tower A to tower C.
Move disk 1 from tower B to tower A.
Move disk 2 from tower B to tower C.
Move disk 1 from tower A to tower C.
Move disk 4 from tower A to tower B.
Move disk 1 from tower C to tower B.
Move disk 2 from tower C to tower A.
Move disk 1 from tower B to tower A.
Move disk 3 from tower C to tower B.
Move disk 1 from tower A to tower C.
Move disk 2 from tower A to tower B.
Move disk 1 from tower C to tower B.
Done in 15 moves.
```

0.7 Fibonacci Sequence

```
In [13]: #include <stdio.h>
         #include <stdlib.h>

         int fib(int n)
         {
             // two base cases for the special values 0,1,1...
             if (n == 0)
             {
                 return 0;
             }
             if (n == 1)
             {
                 return 1;
             }

             // recursive case
             return ( fib(n-1) + fib(n-2) ); // this is the condition for a fibonacci number
         }

         int main()
         {
             int result;

             result = fib(7);

             printf("%d\n", result);

             return EXIT_SUCCESS;
```

```
        }
```

```
13
```

0.8 Program Arguments and `scanf()`

We can pass arguments into `main.c`; it will look like `int main( int argc, char *argv[] )`

- `argc` is the count of the program's arguments
- `argv` is an array of pointers to *character strings* representing the arguments themselves

Say we run `>> myprog left right centre` at the command line:

- `argc`: 4
- `argv`: {"myprog", "left", "right", "centre"}

Note that `argv[0]` will *always* be the name of the program (and is probably not super useful). `argv` is an array of pointers to character arrays, so it is a 2d array of sorts:

- `argv[0]` in the above is "myprog"
- `argv[0][0]` is "m"

0.8.1 `scanf()`

- reads keyboard input
- accepts the same type specifiers as `printf()`
- not on the final exam :)
- annoyingly tricky; probably shouldn't be used in real systems

In [39]:
```c
// scanf doesn't work right in jupyter c kernel

#include <stdio.h>

int main()
{
    int a, b, c;

    printf("Enter the value of a:\n");
    scanf("%d", &a); // toss an int into a, pointerly

    printf("Enter the value of b:\n");
    scanf("%d", &b);
```

```
        printf("Enter the value of c:\n");
        scanf("%d", &c);

        printf("a, b, c: %d, %d, %d\n", a, b, c);
    }
```

```
Enter the value of a:
Enter the value of b:
Enter the value of c:
a, b, c: -1869427792, -1599204144, 32766
```

```
In [40]: #include <stdio.h>

         int main()
         {
             int a, b, c;

             printf("Enter the value of a:\n");
             scanf("%d %d %d", &a, &b, &c); // get 3 whitespace separated integers

             printf("Enter the value of a:\n");
             scanf("%d,%d,%d", &a, &b, &c); // get 3 comma separated integers, with no whitesp

             printf("Enter the value of a:\n");
             scanf("test:%d,%d,%d", &a, &b, &c); // get 3 comma separated integers after the w


             printf("a, b, c: %d, %d, %d\n", a, b, c);
         }
```

Note that in the last case, `scanf` won't read in the integers if they are not preceded by the word `test:`!