Recursive functions

Recursive functions are those that directly (or sometimes indirectly) invoke themselves.

On the *positive side* these functions can produce amazingly succinct solutions to problems that are extremely difficult to solve in other ways. In fact the "other ways" often involve emulation of recursions or transformation of the problem into a more complicated representation.

On the *challenging side* the algorithms can be quite subtle and much more difficult to develop than any of the algorithms that we have seen so far.

The classes of problem for which recursion is most well suited are searches through trees and general graphs.

Applications include:

- parsing of languages that support nested structures (e.g. evaluation of arithmetic expressions in which parentheses may be nested;
- searching for any path or the shortest path through a graph or maze type structure
- generation of subsets of a larger set of elements

Recursion also provides a convenient way to deal with reflected rays in a raytracing environment. However, it is reasonably easy to do this in a non-recursive way as well.

Specifically, when confronted with a multiple alternative path type problem recursion is often a useful approach.
**A simple, but not useful, application**

The "traditional" example of recursion involves computing mathematical recurrences, commonly the factorial function. The factorial (and other recurrences) are trivial to calculate in a single *for* loop in a way that is much more CPU and memory efficient!

Nevertheless, the simple example provides a useful first step in understanding the approach.

```c
#include <stdio.h>

int fact(int val)
{
    if (val == 1)
        return(1);
    val = val * fact(val - 1);
    return(val);
}

int main()
{
    int fv;

    fv = fact(5);
    printf("%d \n", fv);
}
```

If the input value is 1, then *fact* returns 1. Otherwise *fact()* invokes itself and multiplies the value returned by the input value.

The crucial aspect of the procedure is that *no multiplications occur* until 1 is returned.

As described in CPSC 101 each time a function is invoked its parameters and local variables are allocated on the stack. Thus by the time the *return(1)* is reached, there are 5 copies of the parameter *val* on the stack. The values returned by the *return(val)*; are 2, 6, 24, 120.
A version that permits better insight

Because recursive algorithms are more subtle than iterative ones, it is always a good idea to instrument them so that a better understanding of what is transpiring may be obtained. _NEVER_ hesitate to introduce new local variables that can help you see how the computation is progressing... as Professor Brooks said in another context "you will anyway".

```c
#include <stdio.h>

int fact(int val)
{
    int newval;
    int retval;

    if (val == 1)
        return(1);

    newval = fact(val - 1);
    retval = val * newval;

    printf("val = %d newval = %d retval = %d \n", val, newval, retval);

    return(retval);
}

int main()
{
    int fv;

    fv = fact(5);
    printf("%d \n", fv);
}
```

`acad/cs102/examples/fact ==> a.out`

val = 2 newval = 1 retval = 2
val = 3 newval = 2 retval = 6
val = 4 newval = 6 retval = 24
val = 5 newval = 24 retval = 120
120
The proper way to compute a mathematical recurrence

The easy and efficient way requires only a single, simple loop! So don't go looking for ways to use recursion unnecessarily!

```c
#include <stdio.h>

int main()
{
    int fv;
    int i;

    fv = 1;
    for (i = 2; i <= 5; i++)
        fv = fv * i;

    printf("%d \n", fv);
}
```
A more challenging problem

Many problems that are quite easy to state and easy to solve manually can be very difficult to solve with a computer program. Here is one:

*Write all the sequences of the first n letters of the alphabet taken k letters at a time.*

It is well known that the number of combinations of n items taken k at a time is \( \frac{n!}{(n – k)! k!} \)

For example let \( n = 5 \) and \( k = 3 \). Here the number of combinations is \( 5!/ (2! 3!) = (5 \times 4) / 2 = 10 \)

Being clever humans we can easily enumerate them:

We start by writing all the different combinations that start with \( ab \)

\begin{verbatim}
abc
abd
abe
\end{verbatim}

Then we replace the \( b \) with \( c \) and add

\begin{verbatim}
acd
ace
\end{verbatim}

Next \( d \) replaces \( c \) and we get

\begin{verbatim}
ade
\end{verbatim}

That is all there are that contain \( a \) so now we start with \( b \)

\begin{verbatim}
bcd
bce
\end{verbatim}

then

\begin{verbatim}
bde
\end{verbatim}

which leaves us with

\begin{verbatim}
cde
\end{verbatim}

*Exercise:* Try to write a program that will generate these sequences one letter at a time given \( n = 5 \) and \( k = 3 \).
A graphical view of the solution

Recursion depth
1  2  3

Lines shown in --- denote operations done within a loop in the context of a single invocation of the recursive functions.

Lines shown in --- denote a recursive call. Information that must be passed in the recursive call includes

- the current length of the string and
- the base character that the lower layer must start with.

When the current length \( \neq \) target length \( k \), the recursion is finished.
The recursive solution

```c
#include <stdio.h>

/* build string here */
char string[27];
char *alpha = "abcdefghijklmnopqrstuvwxyz";

int n; // total number of characters to work with
int k; // target length of each string

int combos(
    int base,  // index in current string
    int len)  // length of current string
{
    int left = k - len; // remaining items to add

    if (left == 0) {
        printf("%s\n", string);
        return(0);
    }

    /* A single activation of combos will store a character */
    /* only in one spot in the string being constructed. */
    /* That spot is given by len. However the character */
    /* it actually stores in its spot will move one spot */
    /* down the alphabet for each iteration of the loop. */

    while ((base + left) <= n) {
        string[len] = alpha[base];
        string[len + 1] = 0;
        fprintf(stderr, "%3d   %3d   %3d   %3d  %s \n",
            len + 1, base, len, left, string);
        combos(base + 1, len + 1);
        base = base + 1;
    }
    return(0);
}

<table>
<thead>
<tr>
<th>Depth</th>
<th>Base</th>
<th>Len</th>
<th>Left</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>ab</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>abd</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>abe</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>ac</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>acd</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>ace</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>ad</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>ade</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>bc</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>bcd</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>bce</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>bd</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>bde</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>cd</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>cde</td>
</tr>
</tbody>
</table>
```
```c
int main()
{
    string[k] = 0;
    fscanf(stdin, "%d %d", &n, &k);
    combos(0, 0);
}

acad/cs102/examples/combos ==> time a.out | wc -l
26 13
10400600
real    0m5.627s
user    0m1.376s
sys     0m0.152s

int combos(
    int base,     // index in current string
    int len)      // length of current string
{
    int left = k - len;  // remaining items to add

    while ((base + left) <= n) {
        string[len] = alpha[base];
        if (len == (k - 1))
            printf("%s\n", string);
        else
            combos(base + 1, len + 1);

        base = base + 1;
    }
    return(0);
}
```
A maze problem

Consider the following two dimensional array

0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0

We can view this as a maze in which passage through locations have a value of 0 is permitted but passage through locations having a value of 1 is not. We add the additional constraint that movement is permitted between adjacent locations in the array in only vertical and horizontal (not diagonal) directions. As is conventional, the value $\text{maze}[0][4]$ represents row 0 column 4.

Our mission will be given a start (row, col) and a target (row, col), print a path (if one exists) between the start and the finish. For the maze shown here given a start (0, 0) and a target (5, 5) the path is:

0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
Sample input and output

The input will be given in the following format:

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

<table>
<thead>
<tr>
<th>start (row, col)</th>
<th>target (row, col)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>5 5</td>
</tr>
</tbody>
</table>

Maze data. We will always work with 6 x 6 arrays

The path should be printed in the following format. The first 0 (0, 0 )

```
1 (0, 1 )
2 (0, 2 )
3 (0, 3 )
4 (0, 4 )
5 (0, 5 )
6 (1, 5 )
7 (2, 5 )
8 (2, 4 )
9 (2, 3 )
10 (2, 2 )
11 (2, 1 )
12 (2, 0 )
13 (3, 0 )
14 (4, 0 )
15 (4, 1 )
16 (4, 2 )
17 (4, 3 )
18 (5, 3 )
19 (5, 4 )
20 (5, 5 )
```
# 2-d maze search program

```cpp
#include <iostream>
using namespace std;
#define X_DIM 6
#define Y_DIM 6

struct coord_type {
    int y;
    int x;
} path [Y_DIM * X_DIM];

int maze[Y_DIM][X_DIM];
int visited[Y_DIM][X_DIM];

int starty, startx; // (y, x) start coordinates
int targety, targetx; // (y, x) target coordinates
int truelen; // length of the final path

void read_maze(void) {
    int i = 0;
    int *loc = maze[0];

    while (i < X_DIM * Y_DIM) {
        cin >> *loc;
        loc += 1;
        i += 1;
    }
    cin >> starty >> startx;
    cin >> targety >> targetx;
}
```

Because recursion involves a considerable amount of function call/return activity, overhead can be minimized considerably by making global those variables not required for the recursion to work.
The *legal_move()* function

Since we have to evaluate the possibility of moving East, South, West, and North, it is best to write a single function that will return *true* if a potential move is legal and *false* if it is not.

```c
int legal_move(
    int desty,     // potential destination for next step
    int destx)
{
    if  (desty, destx)  is
        • in the maze and
        • hasn't been previously visited
        • has a value of 0
        • return(1);
    else
        return(0);  // false -> not legal move
}
```
The build_path() function

The build_path() function is the recursive function that actually solves the problem. The variable pathlen carries the current depth of the recursion which is also the current length of the path.

Because these algorithms are very subtle it is even for even an experience gdb user to have trouble keeping track what is going on and going wrong. Thus it is a good idea to put diagnostic prints at all entry and exit points of the recursive routine.

Return values are key to making this work. The function must return 0 if

- it discovers that the end of the path is reached or
- it receives a 0 return from a recursive call.

It must return -1 if

- it reaches a point where all moves are illegal
- it is returned -1 all attempts to make legal moves

```c
int build_path(  
    int pathlen,  // current length of path  
    int y,        // (y, x) coordinates of this location  
    int x)  
{
    int rc = -1;  // remember this location has been visited

    fprintf(stderr, "visiting %d %d with pathlen %d \n", y, x, pathlen);

    if this location is the target
    {
        fprintf(stderr, "found the target at %d %d \n", y, x);
        remember current pathlen in truelen
        store this location in the path
        return(0);
    }
}
```
/* Haven't reached the target so need to press onward */

try to build_path() east;
if that doesn't work try to build_path() south;
if that doesn't work try to build_path() west;
if that doesn't work try to build_path() north;

If the target was found, then build_path() returns a non-negative number and the search is over.

Note that the path is built backward as the recursion unwinds.

{  fprintf(stderr, "adding point %d %d with pathlen %d \n",
      y, x, pathlen);
      store this location in the path
      return(0);
}

/* Hit a dead end... back up one spot */
if nothing worked
{
      fprintf(stderr, "stuck at %d %d backing up \n", y, x);
      return(-1)
}

int main()
{
  int failed = 0;
  read_maze();

  failed = build_path(0, starty, startx);
  if (!failed)
    print_path(truelen);
  else
    printf("can't get there from here! \n");
}
visiting 5 4 with pathlen 0
visiting 5 5 with pathlen 1
stuck at 5 5 backing up
visiting 4 4 with pathlen 1
visiting 4 3 with pathlen 2
visiting 4 2 with pathlen 3
visiting 5 2 with pathlen 4
visiting 5 1 with pathlen 5
visiting 5 0 with pathlen 6
visiting 4 0 with pathlen 7
visiting 3 0 with pathlen 8
visiting 3 1 with pathlen 9
visiting 2 1 with pathlen 10
visiting 2 2 with pathlen 11
visiting 2 3 with pathlen 12
visiting 2 4 with pathlen 13
visiting 2 5 with pathlen 14
stuck at 2 5 backing up
stuck at 2 4 backing up
stuck at 2 3 backing up
visiting 1 2 with pathlen 12
visiting 0 2 with pathlen 13
visiting 0 3 with pathlen 14
visiting 0 4 with pathlen 15
visiting 0 5 with pathlen 16
found the target at 0 5