Recursive Functions

1E3
Topic 17

Objectives

- This topic should allow students to
  - Understand a recursive function.
  - Develop recursive solutions to simple problems.
  - Write recursive functions for simple problems.
- This topic is covered in Chapter 16 of the textbook.

Introduction

- "To iterate is human; to recurse is divine"
  - L. Peter Deutsch
- For many problems the most elegant way to solve them is using recursion.
- Recursion solves problems by first solving slightly simpler problems whose solution can be easily turned into a solution of the bigger problem.

Factorial

- Mathematical definition of factorial
  - 0! = 1
  - n! = n x (n-1)!   if n>0
- Use this definition to compute 3!
- It works because each time you use the second equation, you have a smaller problem to solve, and the first equation solves the "smallest" case.

Recursive algorithms

- Where something is solved by solving a smaller version of itself.
- Every recursive algorithm
  - Must have one (or more) **base** cases
    - that is a non-recursive case
  - General case must reduce towards a base case
- Recursive algorithms are implemented using recursive functions
  - Functions that call themselves

Factorial recursive function

```
int fact (int n)
{
  if (n == 0)
      return 1;
  else
      return n * fact(n-1);
}
```


Execution

- Let's see what happens if we execute
  - cout << fact (4);
- Let each of you act as a copy of fact
  - When called on with a parameter (n), execute the body of fact.
  - Call on your next neighbour to compute fact(n-1).
  - When he gives you the answer, multiply it by n and pass it back to whoever called you.
- Also see Fig 16.1 of the text

Mystery function

```
int f (int n, int m)
{
    if (n == 0) return m;
    else
          return f(n-1, m) + 1;
}
```

What is f(4,5)? What is f?

Developing recursive algorithms

- It helps to be lazy.
  - Think of someone else solving a slightly simpler version of my problem.
- Not all "simplifications" will work
  - I need to identify a simpler version of my problem, whose solution I can easily turn into a solution to my bigger problem.

Recursive sum_array

- int sum_array (double a[], int asize)
- A simpler problem is the sum of the first (asize-1) elements of a.

	0	1	2	3	4	5	6	
sum	2	5	6	3	2	8	1	is

	0	1	2	3	4	5		6
sum	2	5	6	3	2	8	+	1

Sum_array base case

- If the general case is
  - sum-array(a, asize) is
    - sum-array(a, asize-1) + a[asize-1]
- What's the base case?
  - In other words when do we not want to apply the general case?
- When asize is 1
  - return a[0]

sum_array code

```
double sum_array (double a[], int size)
{
   if (size == 1)
      return a[0];
   else
      return (sum_array (a, size-1) + a[size-1]);
};
```

See recursivesumsequence.cpp
Compare with iterative version in sumsequence.cpp

Palindromes

- We want to write a function
  - bool is_palindrome (string s);
- Examples of palindromes
  - rotor
  - racecar
  - Madam, I'm Adam (if we remove spaces, punctuation, uppercase letters!)
- Identify a smaller string whose "palindromity" would be helpful.

Palindrome testing

- A string is a palindrome if
  - Its first and last characters match and
  - The rest of the string is a palindrome
- What's the base case?
  - Is "o" a palindrome?
  - Is "" a palindrome?

Palindrome pseudocode

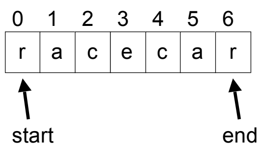
```
is_palindrome (string s) {
  if length(s) <= 1  return true;
  else
      if (first(s) == last(s))
            return (is_palindrome (middle(s));
      else return false;
}
```


Details

- int length(s) => s.length()
- char first(s) => s[0];
- char last(s) => s[s.length() - 1]
- string middle(s) => s.substr(1, s.length() - 2);

Alternative palindrome test

- Our palindrome tester creates a new string, middle, each time around.
- If we think of a string as an array of characters (which it is), we can get a neater, more efficient, test.



if s[0] == s[6] and s[1..5] is a palindrome

then s[0…6] is a palindrome

Recursive helper function

- But the recursive function is a slightly different one
  - It has more parameters

```
bool substr_is_palin (string s, int start, int end);
```

- See palindrome.cpp
- We have to write the original function too:

```
bool is_palindrome (string s) {
   return substr_is_palin (s, 0, s.length()-1); }
```


Is my recursive function correct?

- Like with inductive proofs:
  - Are the base cases correct?
  - Is the general case correct?
  - Does the general case bring the problem closer to a base case?
- If so then by induction, the function will work correctly.
- Check is_palindrome.

What does this do?

```
int f (int a, int b) {
      if (b == 0) return 1;
      else return (a * f(a,b-1));
}
```

e.g. f(2,4)
f(10, 3)

What does this do?

```
int g (int x, int y) {
    if (x == y)
        return 0;
    else return g(x-1, y) + 1;
}
```

e.g. g(4,1)
g(7, 3)

What does this do?

```
int h (double a[], int size) {
    if (size == 0) return 0;
    else if (a[size-1] > 0)
            return h (a, size-1) + 1;
        else return h (a, size-1);
}
```

e.g. h([1.4, -15.3, 23.2, -4.2, 13.0], 5)

Recursive multiplication

- Definition of multiplication:
  - m*n = m*(n-1) + m;
  - m*0 = 0;
- Write a *recursive* function
  - int multiply(int m, int n);
- which multiplies m by n using *repeated addition*.

Recursive reverse digits

- Write a function which reverses the digits of an integer.
  - E.g. rev(12345) is 54321
- Suggest the use of a helper function with two parameters, n and r
  - r holds the digits of n that have already been reversed.
  - E.g. when n is 123, r will be 54