Introduction to Recursion

Today’s Class

• Introduction to recursion: factorials and exponentiation

• How recursion works

• Iteration vs. recursion

• Rules for writing recursive functions

• Examples that we will work on together:
  – Printing a vector in reverse order
  – Binary search

We will look at more sophisticated problems on Thursday.

Recursive Definitions of Factorials and Integer Exponentiation

• The factorial is defined for non-negative integers as

\[ n! = \begin{cases} 
  n \cdot (n - 1)! & n > 0 \\
  1 & n == 0 
\end{cases} \]

• Computing integer powers is defined as:

\[ n^p = \begin{cases} 
  n \cdot n^{p-1} & p > 0 \\
  1 & p == 0 
\end{cases} \]

• These are both examples of recursive definitions.
Recursive C++ Functions

C++, like most other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions.

- Here’s the implementation of factorial:

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

- Here’s the implementation of exponentiation:

  ```cpp
  int intpow( int n, int p )
  {
    if ( p == 0 )
      return 1;
    else
    {
      return n * intpow( n, p-1 );
    }
  }
  ```
The Mechanism of Recursive Function Calls

- When it makes a recursive call (or any function call), a program creates an *activation record* to keep track of
  - The function’s own **completely separate instances** of parameters and local variables.
  - The location in the calling function code to return to when the function is complete.
  - Which activation record to return to when the function is done.

- This is illustrated in the following diagram of the call `fact(4)`. Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns.
• We will draw activation records to illustrate the behavior of other recursive functions as well.
Iteration vs. Recursion

- Each of the above functions could also have been written using a for loop, i.e. iteratively.

- For example, here is an iterative version of factorial:

```c
int ifact( int n )
{
    int result = 1;
    for ( int i=1; i<=n; ++i )
        result = result * i;
    return result;
}
```

- Iterative functions are generally faster than their corresponding recursive functions. Compiler optimizations sometimes (but not always!) can take care of this by automatically eliminating the recursion.

- Sometimes writing recursive functions is more natural than writing iterative functions, however. Most of our examples will be of this sort.
Rules for Writing Recursive Functions

Here is an outline of five steps I find useful in writing and debugging recursive functions:

1. Handle the base case(s) first, at the start of the function.

2. Define the problem solution in terms of smaller instances of the problem. This defines the necessary recursive calls. It is also the hardest part!

3. Figure out what work needs to be done before making the recursive call(s).

4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation.

5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!
Example: Printing the Contents of a Vector

The following example is important for thinking about the mechanisms of recursion.

- Here is a function for printing the contents of a vector. Actually, it is two functions, driver function, and a true recursive function.

```
template <class T>
void print_vec( vector<T>& v )
{
    print_vec( v, 0 );
}

template <class T>
void print_vec( vector<T>& v, unsigned int i )
{
    if ( i < v.size() ) {
        cout << i << ":: " << v[i] << endl;
        print_vec( v, i+1 );
    }
}
```

- What will this print when called in the following code?

```
int main()
{
    vector<int> a;
    a.push_back( 3 ); a.push_back( 5 ); a.push_back( 11 );
    a.push_back( 17 );
    print_vec( a );
}
```

- Note: the idea of a “driver function” that just initializes a recursive function call is quite common.

- How can you change the second `print_vec` function as little as possible to write a recursive function to print the contents of the vector in reverse order?
Binary Search

• Suppose you have a vector<T> v, sorted so that

\[ v[0] \leq v[1] \leq v[2] \leq \ldots \]

• Now suppose that you want to find if a particular value x is in the vector somewhere.

• How can you do this without looking at every value in the vector?

• The solution is an algorithm called binary search. We will write a recursive version in lecture.

Looking Ahead to Thursday

• Every one of the problems we have addressed here can be easily solved using non-recursive techniques.

• This will change substantially on Thursday when we look at two more interesting problems:
  – Sorting a vector using the mergesort technique.
  – Generalizing the word search problem to allow non-straight paths in finding words.