Data Structures — Lecture 3
Order Notation and Recursion

1 Overview

• The median_grade.cpp program from Lecture 2 and background on constructing and using vectors.
• Algorithm analysis; order notation
• Recursion

The material on the order notation at the end the lecture is covered in Ford&Topp, pages 128-139.

2 Algorithm Analysis: What, Why, How?

• What?
  – Analyze code to determine the time required, usually as function of the size of the data being worked on.

• Why?
  – We want to do better than just implementing and testing every idea we have.
  – We want to know why one algorithm is better than another.
  – We want to know the best we can do. (This is often quite hard.)

• How? There are several possibilities:
  1. Don’t do any analysis; just use the first algorithm you can think of that works.
  2. Implement and time algorithms to choose the best.
  3. Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
  4. Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

• What happens in practice?
  – 99% of the time: rough count similar to #4 as a function of the size of the data. Use order notation to simplify the resulting function and even to simplify the analysis that leads to the function.
  – 1% of the time: implement and time.

What follows is a quick review of counting and the order notation.
2.1 Exercise: Counting Example

Suppose \texttt{foo} is an array of \texttt{n} doubles, initialized with a sequence of values.

- Here is a simple algorithm to find the sum of the values in the vector:

  ```
  double sum = 0;
  for ( int i=0; i<n; ++i )
    sum += foo[i];
  ```

- How do you count the total number of operations?
- Go ahead and try. Come up with a function describing the number of operations.
- You are likely to come up with different answers. How do we resolve these differences?

2.2 Order Notation

The following discussion emphasizes intuition. That’s all we care about in Data Structures. For more details and more technical depth, see any textbook on data structures and algorithms.

- Definition

  Algorithm \textit{A} is order \textit{f(n)} — denoted \textit{O(f(n))} — if constants \textit{k} and \textit{n_0} exist such that \textit{A} requires no more than \textit{k \times f(n)} time units (operations) to solve a problem of size \textit{n \geq n_0}.

- As a result, algorithms requiring 3n + 2, 5n – 3, 14 + 17n operations are all \textit{O(n)} (i.e. in applying the definition of order notation \textit{f(n) = n}).

- Algorithms requiring \textit{n^2/10 + 15n – 3} and 10000 + 35n^2 are all \textit{O(n^2)} (i.e. in applying the definition of order notation \textit{f(n) = n^2}).

- Intuitively (and importantly), we determine the order by finding the asymptotically dominant term (function of \textit{n}) and throwing out the leading constant. This term could involve logarithmic or exponential functions of \textit{n}.

- Implications for analysis:
  - We do not need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We do not produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.

- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.
2.3 Common Orders of Magnitude

Here are the most commonly occurring orders of magnitude in algorithm analysis.

- $O(1)$: Constant time. The number of operations is independent of the size of the problem.
- $O(\log n)$: Logarithmic time.
- $O(n)$: Linear time
- $O(n \log n)$
- $O(n^2)$: Quadratic time. Also, Polynomial time
- $O(n^2 \log n)$
- $O(n^3)$: Cubic time. Also, Polynomial time
- $O(2^n)$: Exponential time

2.4 Significance of Orders of Magnitude

- On a computer that performs $10^8$ operations per second:
  - An algorithm that actually requires $15n \log n$ operations requires about 3 seconds on a problem of size $n = 1,000,000$, and 50 minutes on a problem of size $n = 100,000,000$.
  - An algorithm that actually requires $n^2$ operations requires about 3 hours on a problem of size $n = 1,000,000$, and 115 days on a problem of size $n = 100,000,000$.

- Thus, the leading constant of 15 on the $n \log n$ does not make a substantial difference. What matters is the $n^2$ vs. the $n \log n$.
- Moreover, in practice the leading constants usually do not vary by a factor of 15.

2.5 Back to Analysis: A Slightly Harder Example

- Here’s an algorithm to determine if the value stored in variable $x$ is also in an array called $\text{foo}$

```cpp
int loc=0;
bool found = false;
while ( !found && loc < n )
{
    if ( x == foo[loc] )
        found = true;
    else
        loc ++ ;
}
if ( found ) cout << "It is there!\n";
```

- Can you analyze it? What did you do about the if statement? What did you assume about where the value stored in $x$ occurs in the array (if at all)?
2.6 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size vector, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.

- The last is the most common. The first is rarely used.

- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

2.7 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the “size” of the problem.
  - For arrays and other “container classes” this will generally be the number of values stored.

- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations, such as comparisons.

- Do the count, using order notation to describe the result.

2.8 Examples: Loops

In each case give an order notation estimate as a function of $n$ which here notes the

- Version A:

```c
int count=0;
for ( int i=0; i<n; ++i )
  for ( int j=0; j<n; ++j )
    ++count;
```

- Version B:

```c
int count=0;
for ( int i=0; i<n; ++i )
  ++count;
for ( int j=0; j<n; ++j )
  ++count;
```

- Version C: 

```cpp
test count=0;
for ( int i=0; i<n; ++i )
  for ( int j=i; j<n; ++j )
    ++count;
```

• How many operations in each?

2.9 More “O” Examples

Solutions to these will be posted on-line:

1. Write a C++ function to remove the first item in an array of $n$ float values. Give an “O” analysis of this function.

2. Can you analyze binary search? Assume that the search interval is of size $n = 2^k$ for some positive integer $k$.

3 Recursion: The Basics

3.1 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.

3.2 Recursive C++ Functions

C++, like 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:

```c
int intpow( int n, int p )
{
    if ( p == 0 )
        return 1;
    else
    {
        return n * intpow( n, p-1 );
    }
}
```

3.3 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
  - Each newly-called function’s own completely separate instances of parameters and local variables.
  - The location in the calling function code to return to when the newly-called 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.
3.4 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.

3.5 Exercise

Write an iterative version of `intpow`.

3.6 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)!

3.7 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.
void print_vec( vector<int>& v )
{
    print_vec( v, 0 );
}

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

• Exercise: 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.

• Exercise: 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?

3.8 Binary Search

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

    v[0] <= v[1] <= v[2] <= ...

• 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 a recursive algorithm called binary search, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison:
// Here is the recursive function. The "invariant" is that if x is
// in the vector then it must be located within the subscript range
// low to high. Therefore, when low and high are equal, their common
// value is the only possible place for x. Otherwise, the middle
// value is checked, and the search continues recursively in either
// the lower or upper half of the vector.

bool binsearch( const vector<double>& v, int low, int high, double x )
{
    if ( high == low ) return x == v[low];

    int mid = (low+high) / 2;
    if ( x <= v[mid] )
        return binsearch( v, low, mid, x );
    else
        return binsearch( v, mid+1, high, x );
}

// The driver function. It Establishes the search range for the
// value of x based on the minimum and maximum subscripts in the
// vector.

bool binsearch( const vector<double>& v, double x )
{
    return binsearch( v, 0, v.size()-1, x );
}

3.9 Exercises

1. Write a non-recursive version of binary search.

2. If we replaced the if-else structure inside the recursive binsearch function (above) with

   if ( x < v[mid] )
       return binsearch( v, low, mid-1, x );
   else
       return binsearch( v, mid, high, x );

   would the function still work correctly?

4 Summary

- Algorithm analysis and order notation
Recursion is a way of defining a function and or a structure in terms of simpler instances of itself. While we have seen simple examples of recursion here, ones that are easily replaced by iterative, non-recursive functions, later in the semester when we return to recursion we will see much more sophisticated examples where recursion is not easily removed.

On Thursday we will proceed to C++ classes.