CSCI 262 Lecture 10 – Analysis of Recursive Algorithms

Outline

- Analyzing recursive algorithms:
  - How much work is done in the recursive procedure, *exclusive* of the recursive call(s)?
  - How many *calls* are made to the recursive procedure?
  - Multiply the answers to the above two questions!
- Many algorithms have costs that include a $\log_2 n$ factor.
  - $\log_2 n$ is (approximately) the number of times you can divide a number by 2 before reaching 1
  - In Big O we can (usually) ignore the log base – they are all a constant conversion from each other
- Binary search – recursive algorithm to search sorted lists of values in O(log n) time
- Merge sort – recursive algorithm to sort in O(n log n) time

Readings

It's a bit dense, but this article on recursion in ALGOL 60 is kind of fascinating:

https://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/

Essentially, before ALGOL 60, procedural languages (such as FORTRAN) did not implement recursion. Their runtime code did not include a function call stack as we are used to today – it was unnecessary, because every function could have its own "activation record" (the memory used by all the local variables and such – now equivalent to a "stack frame") pre-allocated in memory at compile time – because there was no recursion. Recursion was thus a highly controversial inclusion to ALGOL 60 – the language that went on to be the forebear of just about every major language today (although language family trees are complex, and arguably LISP would have given us recursion eventually).

It's also kind of cool that ALGOL 60 and recursion coincided with the creation of Quicksort, another divide & conquer algorithm for sorting (and for a long time, considered the fastest sort algorithm). The algorithm was devised by Tony Hoare in 1959, but it was the release of ALGOL 60 that let him implement it simply using recursion. We don't study Quicksort in this class, only because the analysis is a bit tricky, but it is O(n log n), and typically much faster than Merge Sort.

https://en.wikipedia.org/wiki/Quicksort#History

Self Check

1. How many times can you divide a number by some number *b* before getting to 1 or less (in a Big O sense)?
2. What would the algorithmic complexity for Merge Sort be if it split into 3 lists instead of 2? (Hint: first figure out how much work `merge()` would need to do, then look at the call tree.)

For Further Practice

1. Extend the `power()` function to work with negative exponents as well as non-negative ones.
2. Analyze the following C++ functions (Big-O, in terms of n) (on back). Be sure you are giving the complexity of the function – how many operations it performs as a function of n – and not the result returned by the function.

a)
```
unsigned int compute(unsigned int n) {
    if (n == 0) return 1;
    else return 1 + compute(n / 2);
}
```

b)
```
unsigned int calculate(unsigned int n) {
    if (n <= 1) return 1;
    else return 1 + calculate(n – 1);
}
```

c)
```
int cipher(unsigned int n) {
    int ans = 0;
    int test = 1;
    while (test < n) {
        test = test * 2;
        ans++;
    }
    return ans;
}
```

d)
```
int count(unsigned int n) {
    if (n == 0) return 0;
    int ans = 0;
    for (int j = 0; j < n; j++) {
        ans++;
    }
    return ans + count(n – 1);
}
```