Module 14: Recap and Some Final Thoughts

Objectives:

1. Review a few of the more confusing aspects related to Big-Oh analysis, recursion, and choice of data structures.
2. Gain some experience analyzing specific problems, and selecting the most appropriate data structures to use for those problems.

Textbook:

No textbook reading.

Assignment:

Complete the assignment for Module 14.

Lab:

No lab for this module.

Lecture:

The purpose of this module is to recap a few of the potentially more confusing topics presented in the course, focusing on a few "big picture" aspects that are not always easy to understand the first time around.

Big-Oh Analysis

With Big-Oh analysis, wWhat we ultimately want to accomplish when we design an algorithm is to strike a balance between minimizing the number of "steps" and minimizing space consumption, with some exceptions. This minimalist approach can thus be summarized as follows:

1. Don't perform any step that is unnecessary. The corollary also holds: all steps that are performed should be necessary.
2. Only declare the minimum number of variables, instantiate the minimum number of classes, etc., needed to perform a task.

When thinking about the number of steps that an algorithm must perform, it is necessary to look at an exploded view of the algorithm. In other words, all compound steps must be expanded, until you are left with a series of atomic steps; i.e., they can be broken down no further. One example of a seemingly simple step is the comparison of two items. We saw item comparisons used in both searching and sorting algorithms. Most examples involving comparisons use integers, since this makes the examples easy to follow. Integer comparisons are simple enough, in fact, that a single comparison equates to a single statement involving one of the equality or inequality operators (==, <, >, <=, >=). However, many comparisons in real world applications are more complex than simply comparing two integers. An example is a file synchronization application that must perform bytewise comparisons between files to allow files on one device to be updated based on the files present on a second device. In this example, the worst case comparison is a comparison between two files of the same length that are equal; i.e., they both have exactly the same contents. In this case, a comparison would entail comparing all

corresponding pairs of bytes in both files. Non-worst case scenarios would include two files that are not the same length; or two files that are the same length, but differ in one or more bytes prior to the last byte. The point here is that a single comparison that may be initiated by a seemingly simple statement that calls a "compare()" method, may actually involve many steps. Since some operations, such as comparisons, may be relatively complex and time-comsuming, that is why the running times of some algorithms are expressed in terms of the number of comparisons that must be made, rather than the total number of statement executions. The number of steps involved in the comparison effectively renders the running time of the other statements in the algorithm as negligible.

On a related note, minimizing the number of steps does not necessarily equate to minimizing the number of lines of code. It is tempting to believe that cramming multiple assignments and operations in a single line of code makes for more efficient code. This is not the case. Condensing code in this way does not reduce the number of instructions that are performed, and if overdone, can lead to loss of readability. In *software development* (as distinguished from *programming*), code readability is an extremely siginificant factor when software must be maintained, as when fixing bugs and extending functionality.

Algorithm analysis can be performed at either the source code level or at the machine code level. Usually we tend to focus more on source code, since that is what is most often generated when developing software. Machine code generation is almost always relegated to a compiler. The machine code that is generated by one compiler will not always be the same as the machine code generated by another compiler, however, and many compilers can have their optimization settings tweaked. For example, a compiler may inline a method call (i.e., copy the entire code for a method in place of using a call to the memory address where the method is stored). Because of this potential disconnect between source code and the corresponding machine code, any analysis of performance based on source code is really only an estimate of the true performance. To precisely determine the efficiency of an algorithm, it is necessary to perform Big-Oh analysis at the machine code level. Fortunately, however, the discrepancy between the source code and the machine code is unlikely to shift performance by an entire order of magnitude, meaning an analysis of the source code will probably suffice in most cases.

Recursion vs. Iteration

All of you covered iteration back in CS I & II, and we covered recursion in this course, focusing on recursion in one module, and seeing it used at various times in later modules. The main advantage of using recursion is that it works extremely well for problems that are recursive in nature. Recursion tends to provide very elegant, minimal code for those problems, while iterative solutions tend to provide code that is not so elegant and much more verbose. The minimal code required by recursion fits very nicely with the minimalist philosophy towards alorithm design, except for a couple of pitfalls. First, if care is not taken, a recursive algorithm can quickly reach exponential running times by performing redundant computations. We saw an example of this with the recursive Fibonacci algorithm. Although the technique of dynamic programming can be used to offset this, dynamic programming requires additional space to store previously computed values. Second, the stack space used to store the activation records of recursive calls is limited, which limits the depth to which recursion can reach. An exception to this is *tail recursion*, a special case of recursion where there are no instructions following the recursive call. With tail recursion, it is possible to reuse the current activation record instead of further deepening the stack by adding new activation records. Some compilers, in fact, are capable of transforming tail recursion into a loop. Whether or not either of these approaches can be used cannot always be controlled, however, since this decision is usually made by the compiler.

The big question then, is, when should recursion be used? As with most questions of this nature, the answer isn't always clear-cut. Some general guidelines that prove helpful are:

1. If building an iterative solution to a problem is too difficult, use a recursive solution, at least for the time being. If an iterative solution can be worked out later, it can be swapped with the recursive solution.

2. If you know the number of recursive calls will be small enough to avoid overflowing the stack space, a recursive solution might be appropriate. Keep in mind, though, that a recursive solution can still take a while to hit the base case, with the call stack fluctuating in size as the recursive algorithm proceeds. One of the Fibonacci labs I presented back in Module 2 illustrates how the number of records on the call stack can fluctuate.

3. When you write a recursive algorithm, always try to check for the base cases first, and the recursive cases last. If possible, write the algorithm so that it employs tail recursion. If the code is compiled by a compiler that is capable of optimizing the tail recursion, it allows you to write the algorithm recursively, yet get better than recursive performance. If the base cases are checked last, tail recursion is impossible.

Selection of Appropriate Data Structures

We covered a number of different data structures during this course, which leads us to revisit the question of why to avoid using simple arrays, or lists, for storing collections of items, and extend it with the question of which data structure to use for a particular problem. Let's discuss the first of those questions: why use anything other than arrays or lists? It should be relatively easy to see why we might want to avoid arrays:

1. They require contiguous blocks of memory, which can pose problems if memory becomes fragmented.

2. Arrays are static in terms of size. If you need to add more items than will fit in the array, you have to resize the array.

3. There are no iterators for arrays, which means that often the only way to traverse an array is to set up an indexed loop, which is prone to off-by-one errors.

In defense of arrays, contiguous blocks of memory are less problematic now that computers have much greater memory capacities than they used to have. Still, if you are writing software for a device with limited memory, you can still have availability problems with contiguous memory. Resizing an array can be relegated to a static, reusable method, so that you don't have to worry about writing this code every time you need to resize an array. As far as array traversals go, programmers are keenly aware of the off-by-one error and are better at avoiding it. Java has also alleviated this problem with its new for-each syntax.

Lists would seem to be ideal, all-purpose data structures. They are dynamic, meaning items can be freely added and removed without the need for resizing an array. Array resizing is done for you for array-based lists, and is not applicable to linked lists. Linked lists eliminate the problems associated with contiguous memory requirements, since each node in the list can point to a neighboring node anywhere in memory. Lists also have iterators that enable clean traversals using a consistent interface.

So why, then, bother with any structure beyond lists? The two major reasons are:

1. Some data structures are specially designed to be used for specific types of applications, which are not possible (or at least not as easily done) with general purpose lists.

2. Strict enforcement of rules that govern how items may be accessed is important.

First of all, some of the data structures we discussed have special structures that enable them to be used for specific applications. Binary search trees, for example, enable us to store items in an ordered way that inherently incorporates binary search capabilities. While the binary search algorithm can be used quite effectively on an ordered array-based list, it is not as effective with a linked list, since retrieving a given node in a linked list by its position has a worst case time of O(N/2), which is significantly greater than the O(logN) time for an array-based list. Higher order trees, which involve greater branching and are thus shorter than binary trees, can provide even faster running times for searches and retrievals. The B tree is specially desgined to deal with data stored outside of main memory, for example on a hard drive. The structure of a B tree is tailored to the block size of the hard drive being used, which minimizes the number of expensive disk access operations.

The graph data structure is also unique in the way ther vertices can be connected to each other. Graphs are basically non-structured, rootless, trees. Any vertex can be the starting point of a traversal or search, and the hierarchy of the vertices in a graph depend on which vertex is chosen as the starting point. Graphs also allow cyclical connections, which trees forbid. None of the shortest path algorithms we discussed will work on any structure other than a graph.

The priority queue data structure is highly specialized because of its reliance on the underlying binary heap. The binary heap, as we discussed, can be implemented as an array by virtue of the fact that a binary heap must be a complete tree. A complete tree ensures there will be no gaps in the tree, and

allows any node to be accessed in O(1) time by utilizing mathematical formulas that only work with complete trees. (A full tree is a special case of a complete tree, so whatever is possible with a complete tree is also possible with a full tree).

Data structures that rely on hashing (HashMap, HashSet) are designed to allow, on average, O(1) access of any item. While arrays and array-based lists also allow O(1) random access, they do so only based on position. Hashed structures allow O(1) access based on either the items themselves, or based on keys associated with the items.

This leaves us with the stack, the queue, and the deque, which would seem to have the weakest argument in their favor. The primary distinction of these structures is the way they limit access to items at one end or the other. A secondary distinction is that they are, by definition, not traversable. Array-based lists and doubly-linked lists can both easily function as stacks, queues, or deques. The main argument against using a list in this capacity is that it opens the door for misuse if a library component is designed to use a stack, queue, or deque internally, but the actual implementation uses a list. A developer who uses such a component may, for example, be able to violate the desired restrictions of the component and access an item stored in the middle of the structure. This may or may not be a serious problem, depending on the application, but if a design calls for LIFO mechanics for a collection, and a stack data structure is available, why not use it instead using a list? Perhaps the most confusing thing about using stacks, queues, and deques is that in many cases there may be a predominant need for LIFO ir FIFO mechanics, but also an occasional need for non-LIFO or non-FIFO mechanics. For example, consider the deck of cards in a poker game application. Cards are typically drawn from the top of the deck, and discarded to the bottom of the deck. This obeys FIFO mechanics, suggesting that a queue would be the appropriate structure to use for modeling the deck. But how then, should we deal with shuffling the deck, which requires access to cards between the first and last cards? If we want to strictly enforce the appropriate access rules, we would probably do something like store the cards in a list when shuffling them, then move them into a queue for game play. Granted, moving all the cards to another data structure is a O(N) operation, but for a standard deck of playing cards, O(N) is equivalent to O(52), which is negligible.

A better justification for ensuring the proper access rules are obeyed in the poker example is if we assume the entire deck exists as a component that is part of a toolkit for building games based on a standard deck of playing cards. If the deck is accessible by some function as a list, that function could potentially violate the expected access rules, which could lead to unexpected results. For example, someone could build a poker game that selectively deals cards from the middle of the deck, and could even stack the deck in favor of one player. Users who play the game would have no reason to expect such behavior, and may become disillusioned with the game.

Final Thoughts

This course was designed to be at the CS III level, which corresponds to an introductory course on data structures and algorithms. I've tried to make the course topics conform to those recommended by the ACM Computing Curricula. Although the title of the course sounds all-encompassing, what we have covered really only scratches the surface. There are many other types of data structures we did not have time to explore (mostly tree and graph variants), and there are considerably more algorithms than we had time to cover.

If you have any other questions about any of the topics we have covered this semester, please feel free to email them to me or post them on the discussion forum, and I will do the best I can to answer them for you.