SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Complexity and Asymptotic Analysis

You have studied the "extensible array" abstract data type, known in Java as the `Vector` or `ArrayList`. This structure affords us a meaningful opportunity to look at important efficiency issues before moving on to more complicated and interesting structures and the algorithms that use them.

Consider these observations:

- A programmer can use an `ArrayList` in contexts where an array could be used.
- The `ArrayList` hides some of the complexity associated with inserting or removing values from the middle of the array, or when the array needs to be resized.
- As a user of an `ArrayList`, these potentially expensive operations all seem very simple – it's just a method call.
- But.. programmers who make use of abstract data types need to be aware of the actual costs of the operations and their effect on their program's efficiency.

We will now spend some time looking at how Computer Scientists measure the costs associated with our structures and the operations on those structures.

Costs of `ArrayList` Operations

In class, we will build a table of the key operations of the `ArrayList` ADT and how expensive each is. We will formalize the idea soon, but to start, we will just consider how many array accesses are needed, which is usually highly dependent on the number of times the loops need to iterate to complete the operation's functionality.

- `add` (which by default in an `ArrayList` adds at the end)
- `add` at a position (consider 0 as an important special case)
- `remove` from a position (consider removing the first or last as important special cases)
- `get`/`set`
- `contains`/`indexOf` (consider unsuccessful searches and successful searches that find the value at the beginning, at a position somewhere in the middle, or at the last position)
- `isEmpty`

- `size`
- `clear`

It is also very important to consider that the `add` operations sometimes are called on an `ArrayList` that is already full, meaning the `ensureCapacity` operation will need to make the internal array of the `ArrayList` larger before adding the new element.

The default behavior, both in Java's `ArrayList` and in the version we worked with in lab, is to double the size of the internal array each time it needs to grow.

You might have some concern that this potentially wastes a lot of space. It means that an `ArrayList` to which we add n elements would have an internal array with a size somewhere between n and $2n$.

Another option would be to grow the array by 1 each time it needs to grow, eliminating any unused slots.

So let's consider those options, and for to keep the math manageable, we will assume that our `ArrayList` starts with a capacity of 1 and that we are adding n elements with the default `add` operation (adding to the end), where n is a power of 2.

For the case where we increase the capacity by 1, we need to grow the internal array on every step. For the k^{th} add, we would need to allocate a new array of size k and copy over the $k-1$ elements from the old array on every step.

This will require about $\frac{n^2}{2}$ copy operations:

$$0 + 1 + 2 + 3 + 4 + ... + n = n * \frac{n-1}{2}$$

If we double the array size, many of our add operations will not need any copies at all - they have available space to drop into. Our total number of elements to copy will be

$$0 + 1 + 2 + 4 + 8 + ... + \frac{n}{2} = n - 1$$

Copying about n elements is much less painful than copying $\frac{n^2}{2}$.

Of course, no copies would need to be made if we just allocated space for n elements at beginning (a good idea, if you know n ahead of time, but if you did, you might just be using an array...).

These kinds of differences relate to the tradeoffs made when developing algorithms and data structures. We could avoid all of these copies by just allocating a huge array, larger than we could ever possibly need, right at the start. That would be very efficient in terms of avoiding the work of copying the contents of the array, but it is very inefficient in terms of memory usage.

This is an example of a *time* vs. *space tradeoff*. We can save some time (do less computing) by using more space (less memory). Or vice versa.

We also observe that the cost to add an element to an `ArrayList` is not constant! Usually it is – when the `ArrayList` is already big enough – but in those cases where the `ArrayList` has to be expanded, it involves copying over all of the elements already in the `ArrayList` before adding the new one. This cost will depend on the number of elements in the `ArrayList` at the time.

The cost of inserting or removing an element from the middle or beginning of an `ArrayList` always depends on how many elements are in the `ArrayList` after the insert/remove point.

Asymptotic Analysis

We want to focus on how Computer Scientists think about the differences among the costs of various operations.

There are many ways that we can think about the "cost" of a particular computation. The most important of which are

- *computational cost*: how many *basic operations* of some kind does it take to accomplish what we are trying to do?
  - If we are copying the elements of one array to another, we might count the number of elements we need to copy.
  - In other examples, we may wish to count the number of times a key operation, such as a multiplication statement, takes place.
  - We can estimate running time for a problem of size n, $T(n)$, by multiplying the execution time of our basic operation, c_{op}, by the number of basic operations, $C(n)$:

$$T(n) \approx c_{op}C(n)$$

- *space cost*: how much memory do we need to use?
  - may be the number of bytes, words, or some unit of data stored in a structure

The operations we'll want to count tend to be those that happen inside of loops, or more significantly, inside of nested loops.

Finding the "Trends"

Determining an exact count of operations might be useful in some circumstances, but we usually want to look at the *trends* of the operation costs as we deal with larger and larger problem sizes.

This allows us to compare algorithms or structures in a general but very meaningful way without looking at the relatively insignificant details of an implementation or worrying about characteristics of the machine we wish to run on.

To do this, we ignore differences in the counts which are constant and look at an overall trend as the size of the problem is increased.

For example, we'll treat n and $\frac{n}{2}$ as being essentially the same.

Similarly, $\frac{1}{1000}n^2$, $2n^2$ and $1000n^2$ are all "pretty much" n^2.

With more complex expressions, we also say that only the most significant term (the one with the largest exponent) is important when we have different parts of the computation taking different amounts of work or space. So if an algorithm uses $n + n^2$ operations, as n gets large, the n^2 term dominates and we ignore the n.

In general if we have a polynomial of the form $a_0n^k + a_1n^{k-1} + ... + a_k$, say it is "pretty much" n^k. We only consider the most significant term.

Defining "Big O" Formally

We formalize this idea of "pretty much" using *asymptotic analysis*:

Definition: A function $f(n) \in O(g(n))$ if and only if there exist two positive constants c and n_0 such that $|f(n)| \leq c \cdot g(n)$ for all $n > n_0$.

Equivalently, we can say that $f(n) \in O(g(n))$ if there is a constant c such that for all sufficiently large n, $|\frac{f(n)}{g(n)}| \leq c$.

To satisfy these definitions, we can always choose a really huge $g(n)$, perhaps n^{n^n}, but as a rule, we want a $g(n)$ without any constant factor, and as "small" of a function as we can.

So if both $g(n) = n$ and $g(n) = n^2$ are valid choices, we choose $g(n) = n$. We can think of $g(n)$ as an upper bound (within a constant factor) in the long-term behavior of $f(n)$, and in this example, n is a "tighter bound" than n^2.

We also don't care how big the constant is and how big n_0 has to be. Well, at least not when determining the complexity. We would care about those in specific cases when it comes to implementation or choosing among existing implementations, where we may know that n is not going to be very large in practice, or when c has to be huge. But for our theoretical analysis, we don't care. We're interested in *relative rates of growth* of functions.

Common Orders of Growth

The most common *orders of growth* or *orders of complexity* are

- $O(1)$ – for any *constant*-time operations, such as the assignment of an element in an array. The cost doesn't depend on the size of the array or the position we're setting.
- $O(\log n)$ – *logarithmic* factors tend to come into play in "divide and conquer" algorithms. Example: binary search in an ordered array of n elements.
- $O(n)$ – *linear* dependence on the size. This is very common, and examples include the insertion of a new element at the beginning of an array containing n elements.
- $O(n \log n)$ – this is just a little bigger than $O(n)$, but definitely bigger. The most famous examples are divide and conquer sorting algorithms, which we will look at soon.

- $O(n^2)$ – *quadratic*. Most naive sorting algorithms are $O(n^2)$. Doubly-nested loops often lead to this behavior. Example: matrix-matrix addition for $n \times n$ matrices.
- $O(n^3)$ – *cubic* complexity. Triply nested loops will lead to this behavior. A good example is "naive" matrix-matrix multiplication. We need to do n operations (a dot product) on each of n^2 matrix entries.
- $O(n^k)$, for constant k – *polynomial* complexity. As k grows, the cost of these kinds of algorithms grows very quickly.

  Computer Scientists are actually very excited to find polynomial time algorithms for seemingly very difficult problems. In fact, there is a whole class of problems (NP) for which if you could either come up with a polynomial time algorithm, no matter how big k is (as long as it's constant), or if you could prove that no such algorithm exists, you would instantly be world famous! At least among us Computer Scientists. We will likely introduce the idea of NP and NP-Completeness near the end of Analysis of Algorithms.
- $O(2^n)$ – *exponential* complexity. Recursive solutions where we are searching for some "best possible" solution often leads to an exponential algorithm. Constructing a "power set" from a set of n elements requires $O(2^n)$ work. Checking topological equivalence of circuits is one example of a problem with exponential complexity.
- $O(n!)$ – *factorial* complexity. This gets pretty huge very quickly. We are already considering one example on the first problem set: traversing all permutations of an n-element set.
- $O(n^n)$ – even more huge

Suppose we have operations with time complexity $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, and $O(2^n)$.

And suppose the time to solve a problem of size n is t. How much time to do problem 10, 100, or 1000 times larger?

Time to Solve Problem				
size	n	$10n$	$100n$	$1000n$
$O(1)$	t	t	t	t
$O(\log n)$	t	$> 3t$	$\sim 6.5t$	$< 10t$
$O(n)$	t	$10t$	$100t$	$1,000t$
$O(n \log n)$	t	$> 30t$	$\sim 650t$	$< 10,000t$
$O(n^2)$	t	$100t$	$10,000t$	$1,000,000t$
$O(2^n)$	t	$\sim t^{10}$	$\sim t^{100}$	$\sim t^{1000}$

Note that the last line depends on the fact that the constant is 1, otherwise the times are somewhat different.

Now let's think about complexity from a different perspective.

Suppose we get a faster computer, 10, 100, or 1000 times faster than the one we had, or we're willing to wait 10, 100, or 1000 times longer to get our solution if we can solve a larger problem.

How much larger problems can be solved? If original machine allowed solution of problem of size k in time t, then how big a problem can be solved in some multiple of t?

Problem Size				
speed-up	1x	10x	100x	1000x
$O(\log n)$	k	k^{10}	k^{100}	k^{1000}
$O(n)$	k	$10k$	$100k$	$1,000k$
$O(n \log n)$	k	$< 10k$	$< 100k$	$< 1,000k$
$O(n^2)$	k	$3k+$	$10k$	$30k+$
$O(2^n)$	k	$k+3$	$k+7$	$k+10$

For an algorithm which works in $O(1)$, the table makes no sense - we can solve as large a problem as we like in the same amount of time. The speed doesn't make it any more likely that we can solve a larger problem.

Examples

- Filling in a difference table, addition table, multiplication table, *etc.*, $O(n^2)$
- Inserting n elements into an `ArrayList` using default `add`, $O(n)$
- Inserting n elements into an `ArrayList` using `add` at position 0, $O(n^2)$

As we saw with our `ArrayList` operations, some ADT operations or algorithms will have varying complexities depending on the specific input. So we can consider three types of analysis:

- *Best case*: how fast can an instance be if we get really lucky?
  - find an item in the first place we look in a search – $O(1)$
  - get presented with already-sorted input in certain sorting procedures – $O(n)$
  - we don't have to expand an `ArrayList` when adding an element at the end – $O(1)$
- *Worst case*: how slow can an instance be if we get really unlucky?
  - find an item in the last place in a linear search – $O(n)$
  - get presented with a reverse-sorted input in certain sorting procedures – $O(n^2)$
  - we have to expand an `ArrayList` to add an element – $O(n)$
- *Average case*: how will we do on average?
  - linear search – equal chance to find it at each spot or not at all – $O(n)$
  - get presented with reasonably random input to certain sorting procedures – $O(n \log n)$

- we have to expand an `ArrayList` sometimes, complexity depends on how we resize and the pattern of additions

Important note: this is **not** the average of the best and worst cases!

Examples

To practice with determining the Big-O complexity of various operations, let's look at some additional examples.

Example 0

```
for(int i = 0; i < n; i++)
{
    //do something
}
```

Example 1

```
for(int i = 0; i < 5 * n; i++)
{
    //do something
}
```

Example 2

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 10; j++)
    {
        //do something
    }
}
```

Example 3

```
for(int i = 0; i < n; i++)
{
    for(int j = 0; j < n; j++)
    {
        //do something
    }
}
```

Example 4

```
for(int i = 0; i < n; i++)
{
    for(int j = 0; j < n * n; j++)
    {
        //do something
    }
}
```

Example 5

```
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= i; j++)
    {
        System.out.println(i + " " + j);
    }
}
```

Example 6

```
/**
  * Precondition:  The array contains unique values sorted ascending.
  * @param arr The input array.
  * @param lookFor The value to look for in the input array.
  * @return The index of the value if found in the array; -1 if not found.
  */
 public static int binarySearch(int[] arr, int lookFor)
 {
     int low = 0;
     int mid = arr.length / 2;
     int high = arr.length - 1;

     while(low <= high){
         if(arr[mid] == lookFor){
             return mid;
         } else if(arr[mid] < lookFor){
             low = mid + 1;
         } else {
             high = mid - 1;
         }
         mid = (low + high) / 2;
```

```
     }
     return -1;
 }
```

Example 7

Here, we say N = `data.length`.

```
for (int j = 5; j < data.length - 2; j++)
{
    data[j] = 0;
}
```

Example 8

```
for (int j = 0; j < 500; j++)
{
    for (int i = 0; i < j; i++)
    {
         if (data[i] == data[j])
             return true;
    }
}
```

Example 9

```
for (int j = 0; j < data.length; j++)
{
    for (int i = 0; i < j; i++)
    {
         if (data[i] == data[j])
             return true;
    }
}
```