SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2024

Topic Notes: Analysis Fundamentals

We will next review and greatly expand upon some of what you know from some of your previous courses about measuring efficiency.

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!

Basic Efficiency Classes

Big O is only one of three asymptotic notations we will use.

Informally, the three can be thought of as follows:

- $O(g(n))$ is set of all functions that grow at the **same rate as** or **slower than** $g(n)$.
- $\Omega(g(n))$ is set of all functions that grow at the **same rate as** or **faster than** $g(n)$.
- $\Theta(g(n))$ is set of all functions that grow at the **same rate as** $g(n)$.

We previously gave the formal definition of $O(g(n))$:

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 $n > n_0$.

Now, let's see/remember how we can use this definition to prove that a function is in a particular efficiency class.

Let's show that

$$500n + 97 \in O(n^2)$$

by finding appropriate constants c and n_0 to match the definition.

Since all we need to do is to produce **any** pair of constants to meet the requirement, we have a great deal of freedom in selecting our constants. We could select very large constants that would satisfy the definition. But we will attempt to obtain some fairly small ("tight") constants.

Note that

$$500n + 97 \leq 500n + n$$

for $n \geq 97$. And

$$500n + n = 501n \leq 501n^2$$

indicating that we can use $c = 501$.

So, $c = 501$ and $n_0 = 97$ will work.

Alternately, we could notice that

$$500n + 97 \leq 500n + 97n$$

for $n \geq 1$. And

$$500n + 97n = 597n \leq 597n^2$$

indicating a value of $c = 597$ to go with $n_0 = 1$.

Similar arguments work for other polynomials.

To show that

$$27n^3 + 12n^2 + 25000 \in O(n^3)$$

We can proceed as follows:

$$27n^3 + 12n^2 + 25000 \leq 27n^3 + 12n^2 + n$$

for $n \geq 250000$. And

$$27n^3 + 12n^2 + n \leq 27n^3 + 12n^3 + n^3 = 40n^3$$

So we can use $c = 40$ and $n_0 = 25000$ to satisfy the definition, showing that $27n^3+12n^2+25000 \in O(n^3)$.

Next, let's work toward a more general result:

$$an^2 + bn + d \in O(n^2)$$

for positive constants a, b, d.

We proceed by noting that

$$an^2 + bn + d \leq an^2 + bn + n$$

for $n > d$, and

$$an^2 + bn + n = an^2 + (b + 1)n \leq an^2 + n^2$$

for $n > b + 1$, and

$$an^2 + n^2 = (a + 1)n^2$$

which leads us to constants of $c = a + 1$ and $n_0 = \max(d, b + 1)$.

Next, we consider the formal definitions of Ω and Θ.

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

Definition: A function $f(n) \in \Theta(g(n))$ if and only if there exist three positive constants c_1, c_2, and n_0 such that $c_2 \cdot g(n) \leq |f(n)| \leq c_1 \cdot g(n)$ for all $n > n_0$.

Similar techniques can be used to prove membership of a function in these classes.

To show that $15n^2 + 37 \in \Omega(n)$, we need to show a lower bound instead of an upper bound as we did for Big-O proofs. So instead of making our function larger to help make progress, we can make our function smaller.

$$15n^2 + 37 \geq 15n^2 \geq 15n$$

where the latter inequality holds for any $n \geq 1$. So we can choose $n_0 = 1$ and $c = 15$ to satisfy the definition.

To show that $\frac{1}{2}n(n-1) \in \Theta(n^2)$, we need to show both the upper and lower bounds hold.

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$

for $n \geq 0$. So for the right inequality (the upper bound), we can choose $c_1 = \frac{1}{2}$ and $n_0 = 0$.

To prove the left inequality, we can observe that

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n$$

when $n \geq 2$, and

$$\frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n = \frac{1}{2}n^2 - \frac{1}{4}n^2 = \frac{1}{4}n^2$$

So for the lower bound, we can choose $c_2 = \frac{1}{4}$ but we need $n_0 = 2$. This gives us, overall, $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

A more straightforward one:

Show that $7n^3 - 4n + 17 \in \Theta(n^3)$.

First, the upper bound proof.

$$7n^3 - 4n + 17 \leq 7n^3 - 4n + 17n$$

when $n \geq 1$, and

$$7n^3 - 4n + 17n = 7n^3 + 13n \leq 7n^3 + 13n^3 = 20n^3$$

Then for the lower bound.

$$7n^3 - 4n + 17 \geq 7n^3 - 4n \geq 7n^3 - 4n^3$$

when $n \geq 1$, and

$$7n^3 - 4n^3 = 3n^3$$

So we let $c_1 = 20$, $c_2 = 3$, and $n_0 = 1$.

Some Useful Properties

As we work with these asympotic notations, the following properties will often prove useful. We will not prove them formally, but convince yourself that these hold (and use them as needed!).

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Using Limits

A powerful means of comparing the orders of growth of functions involves the use of limits. In particular, we can compare functions $f(n)$ and $g(n)$ by computing the limit of their ratio:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Three cases commonly arise:

- 0: $f(n)$ has a smaller order of growth than $g(n)$, *i.e.*, $f(n) \in O(g(n))$.
- $c > 0$: $f(n)$ has the same order of growth as $g(n)$, *i.e.*, $f(n) \in \Theta(g(n))$.
- ∞: $f(n)$ has a larger order of growth than $g(n)$, *i.e.*, $f(n) \in \Omega(g(n))$.

Two rules that often come in handy when using this technique:

L'Hôpital's Rule states

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

and *Stirling's formula* states

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

for large values of n.

Let's consider some examples:

1. Compare $f(n) = 20n^2 + n + 4$ and $g(n) = n^3$.

$$\begin{aligned}\lim_{n\to\infty} \frac{20n^2 + n + 4}{n^3} &= \lim_{n\to\infty} \left[\frac{20n^2}{n^3} + \frac{n}{n^3} + \frac{4}{n^3}\right] \\ &= \lim_{n\to\infty} \left[\frac{20}{n} + \frac{1}{n^2} + \frac{4}{n^3}\right] \\ &= 0 + 0 + 0 = 0\end{aligned}$$

   so f has a slower growth than g, $f(n) \in O(g(n))$.

2. Compare $f(n) = n^2$ and $g(n) = n^2 - n$.

$$\lim_{n\to\infty} \frac{n^2}{n^2 - n} = \lim_{n\to\infty} \frac{n}{n-1} = 1$$

so f and g have the same growth rate.

3. Compare $f(n) = 2^{\log n}$ and $g(n) = n^2$.

$$\lim_{n\to\infty} \frac{2^{\log n}}{n^2} = \lim_{n\to\infty} \frac{n^{\log 2}}{n^2} = \lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

so n^2 grows faster.

4. Compare $f(n) = \log(n^3)$ and $g(n) = \log(n^4)$.

$$\lim_{n\to\infty} \frac{\log(n^3)}{\log(n^4)} = \lim_{n\to\infty} \frac{3\log(n)}{4\log(n)} = \frac{3}{4}$$

so these grow at the same rate.

5. Compare $f(n) = \log_2(n)$ and $g(n) = n$.

$$\lim_{n\to\infty} \frac{\log_2(n)}{n} = \lim_{n\to\infty} \frac{\frac{1}{n\ln 2}}{1} = 0$$

so n grows faster (as we know anyway).

Analyzing Nonrecursive Algorithms

We will next look at how to analyze non-recursive algorithms.

Our general approach involves these steps:

1. Determine the parameter that indicates the input size, n.
2. Identify the basic operation.
3. Determine the worst, average, and best cases for inputs of size n.
4. Specify a sum for the number of basic operation executions.
5. Simplify the sum

Example 1: Finding the Maximum Element

Our first algorithm to analyze is one to search for the maximum element in an array.

ALGORITHM MAXELEMENT(A)
//Input: an array $A[0..n-1]$
$maxval \leftarrow A[0]$
for $i \leftarrow 1..n-1$ **do**
if $A[i] > maxval$ **then**
$maxval \leftarrow A[i]$
return $maxval$

The input size parameter is n, the number of elements in the array.

The basic operation could be the comparison or the assignment in the for loop. We choose the comparison since it executes on every loop iteration.

Since this basic operation executes every time through the loop regardless of the input, the best, average, and worst cases will all be the same.

We will denote the number of comparisons as $C(n)$. There is one comparison in each iteration of the loop, so we can (overly formally) specify the total as:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Example 2: Element Uniqueness Problem

Our next example algorithm is one that determines whether all of the elements in a given array are distinct.

ALGORITHM UNIQUEELEMENTS(A)
//Input: an array $A[0..n-1]$
for $i \leftarrow 0..n-2$ **do**
for $j \leftarrow i+1..n-1$ **do**
if $A[i] = A[j]$ **then**
return $false$
return $true$

Again, the input size parameter n is the number of elements in the array.

The basic operation is the comparison in the body of the inner loop.

The number of times this comparison executes depends on whether and how quickly a matching pair is located. The best case is that `A[0]` and `A[1]` are equal, resulting in a single comparison. The average case depends on the expected inputs and how likely matches are. We do not have enough information to analyze this formally. So we will focus on the worst case, which occurs when there is no match and all loops execute the maximum number of times.

How many times will the comparison occur in this case? The outer loop executes $n-1$ times. For the first execution of the inner loop, the comparison executes $n-2$ times. The second time around, we do $n-3$ comparisons. And so on until the last iteration that executes just once.

So we compute our worst case number of comparisons:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\ &= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \end{aligned}$$

From here, we can factor out the $(n-1)$ from the first summation and apply the second summation rule from Levitin p. 476 to the second summation to obtain:

$$\begin{aligned} C(n) &= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{2(n-1)^2}{2} - \frac{(n-2)(n-1)}{2} \\ &= \frac{n(n-1)}{2} \in \Theta(n^2). \end{aligned}$$

This isn't surprising at all, if we think about what the loops are doing.

Example 3: Matrix Multiplication

Recall the algorithm for multiplying two $n \times n$ matrices:

ALGORITHM MATRIXMULTIPLY(A,B)
    //Input: an array $A[0..n-1][0..n-1]$
    //Input: an array $B[0..n-1][0..n-1]$
    for $i \leftarrow 0..n-1$ **do**
        for $j \leftarrow 0..n-1$ **do**
            $C[i][j] \leftarrow 0$
            for $k \leftarrow 0..n-1$ **do**
                $C[i][j] \leftarrow C[i][j] + A[i][k] * B[k][j]$
    return C

The input size is measured by n, the order of the matrix.

The basic operation could be the multiplication or the addition in the innermost loop. Generally, we would choose the multiplication (it's often a more expensive operation), but since they both

happen the same number of times, it doesn't matter which we pick. We just want to count the number of times that line executes.

The best, average, and worst case behavior are identical: the loops all need to execute to completion.

So we're ready to set up our summation for the number of multiplications:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

We can go a step further and estimate a running time, if the cost of a multiplication on a given machine is c_m.

$$T(n) \approx c_m M(n) = c_m n^3.$$

And this can be extended to include additions (where each of $A(n)$ additions costs c_a.

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3.$$

This is just a constant multiple of n^3.

Example 4: Number of Binary Digits Needed for a Number

We next consider a very different example, an algorithm to determine how many bits are needed to represent a positive integer in binary.

```
binary(n)

  count = 1
  while (n > 1)
    count++
    n = floor(n/2)

  return count
```

Our summation techniques will not work here – while this is not a recursive algorithm, the approach here will involve recurrence relations, which are usually applied to recursive algorithm analysis. So we delay our answer to this one until we have seen the appropriate techniques.

Analyzing Recursive Algorithms

Our approach to the analysis of recursive algorithms differs somewhat. The first three steps are the same: determining the input size parameter, identifying the basic operation, and separating best, average, and worst case behavior.

Setting up a summation is replaced by setting up and solving a *recurrence relation*.

Example 1: Computing a Factorial

We start with a simple recursive algorithm to find $n!$:

ALGORITHM FACTORIAL(n)
  if $n = 0$ **then**
    return 1
  else
    return $n \cdot$ FACTORIAL$(n-1)$

The size is n (the parameter passed in to get things started) and the basic operation is the multiplication in the `else` part.

There is no difference among the best, average, and worst cases.

You are hopefully familiar with recurrence relations from your math experience, but don't worry, we'll talk about how to set them up and to solve them.

The recurrence for this problem is quite simple:

$$M(n) = M(n-1) + 1$$

Why? The total number of multiplications, $M(n)$, to compute $n!$ is the number of multiplications to compute $(n-1)!$, which we can denote as $M(n-1)$, plus the 1 to get from $(n-1)!$ to $n!$.

We do need a *stopping condition* or *base case* for this recurrence, just as we have a stopping condition for the algorithm. For $n = 0$, we do not need to do any multiplications, so we can add the initial condition $M(0) = 0$.

We can easily determine that $M(n) = n$ just by thinking about this for a few minutes. But instead, we will worth through this by using back substitution.

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3 \end{aligned}$$

Our goal is to find the pattern – what would our recurrence look like in terms of some $M(n-i)$?

$$M(n) = M(n-i) + i$$

In order to complete the process, we need to be able to apply the base case. If we choose $i = n$, that will allow us to do so:

$$M(n) = M(n-n) + n = M(0) + n = n \in \Theta(n).$$

Example 2: Towers of Hanoi

Many of you are all likely to be familiar with the Towers of Hanoi.

Recall that solving an instance of this problem for n disks involves solving an instance of the problem of size $n-1$, moving a single disk, then again solving an instance of the problem of size $n-1$. We denote the number of moves to solve the probem for n disks as $M(n)$.

So we have the recurrence:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \\ M(1) &= 1 \end{aligned}$$

Again, we can proceed by backward substitution.

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2^1 + 2^0 \end{aligned}$$

In terms of an arbitrary $M(n-i)$, we can write:

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + ... + 2^2 + 2^1 + 2^0$$

This time to be able to apply our base case, we need to set $i = n-1$, which will give us a formula based on $M(1)$.

$$\begin{aligned} M(n) &= 2^{n-1} M(n-(n-1)) + 2^{n-2} + 2^{n-3} + ... + 2 + 1 \\ &= 2^{n-1} M(1) + 2^{n-2} + 2^{n-3} + ... + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2 + 1 \\ &= 2^n - 1 \in \Theta(2^n). \end{aligned}$$

Example 3: Number of Binary Digits Needed for a Number

We return now to the problem of determining how many bits are needed to represent a positive integer in binary.

To find the recurrence more readily, we recast the problem recursively:

ALGORITHM BINDIGITS(n)
  if $n = 1$ **then**
    return 1
  else
    return BINDIGITS($\lfloor \frac{n}{2} \rfloor$) $+ 1$

In this case, we will count the number of additions, $A(n)$. For a call to this function, we can see that $A(1) = 0$, and

$$A(n) = A(\lfloor n/2 \rfloor) + 1$$

when $n > 1$.

It is important to remember here that we are counting the number of additions, not computing the return value of the function!

The problem is a bit complicated by the presence of the floor function. We can only be precise and apply backward substitution only if we assume that n is a power of 2. Fortunately, we can do this and still get the correct order of growth (by the *smoothness rule*, which we will just take as fact – interested students can find a proof).

So assuming $n = 2^k$, we know that $A(1) = A(2^0) = 0$ and

$$A(2^k) = A(2^{k-1}) + 1$$

for $k > 0$. So we can proceed by backward substitution.

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 \end{aligned}$$

Again, we will look for the pattern, determining a formula based on $A(2^{k-i})$.

$$A(2^k) = A(2^{k-i}) + i$$

This time, to apply our base case defined by the value of $A(2^0)$, we need to set $i = k$.

$$A(2^k) = A(2^{k-k}) + k = A(2^0) + k = k.$$

Since we let $n = 2^k$, we also know that $k = \log_2 n$, so we can convert our result back to be in terms of n:

$$A(n) = \log_2 n \in \Theta(\log n).$$

Example 4: Another recurrence example

Suppose in the analysis of some algorithm, we find the following recurrence:

$$C(n) = 2C(n/2) + 2$$

when $n > 0$, and a base case of $C(1) = 0$.

Again, we will assume $n = 2^k$, so our base case is now $C(1) = C(2^0) = 0$, and our recurrence becomes

$$C(2^k) = 2C(2^{k-1}) + 2$$

Our backward substitution this time proceeds as follows:

$$\begin{aligned}
C(2^k) &= 2C(2^{k-1}) + 2 \\
&= 2[2C(2^{k-2}) + 2] + 2 \\
&= 2[2[2C(2^{k-3}) + 2] + 2] + 2 \\
&= 2^3C(2^{k-3}) + 8 + 4 + 2 \\
&= 2^3[2C(2^{k-4}) + 2] + 8 + 4 + 2 \\
&= 2^4C(2^{k-4}) + 16 + 8 + 4 + 2
\end{aligned}$$

Our general formula in terms of an arbitrary step based on $C(2^{k-i})$ looks like this:

$$C(2^k) = 2^iC(2^{k-i}) + 2^i + 2^{i-1} + ... + 2^2 + 2^1$$

In order to be able to apply our base case, which gives us a value for $C(2^0)$, we need to let $i = k$, giving

$$\begin{aligned} C(2^k) &= 2^k C(2^{k-k}) + 2^k + 2^{k-1} + ... + 2 \\ &= 2^k C(2^0) + 2^k + 2^{k-1} + ... + 2 \\ &= 2^k \cdot 0 + 2^k + 2^{k-1} + ... + 2 \\ &= 2^k + 2^{k-1} + ... + 2 \\ &= \sum_{i=1}^{k} 2^i = [2^{k+1} - 1] - 1 = 2^{k+1} - 2 \end{aligned}$$

Finally, we convert back to be in terms of the original n:

$$= 2 \cdot 2^k - 1 = 2n - 2 \in \Theta(n)$$

Master Theorem

Many of the recurrences here will arise when analyzing *divide and conquer* algorithms, a a very common and very powerful algorithm design technique. The general idea:

1. Divide the complete instance of problem into two (sometimes more) subproblems that are smaller instances of the original.
2. Solve the subproblems (recursively).
3. Combine the subproblem solutions into a solution to the complete (original) instance.

While the most common case is that the problem of size n is divided into 2 subproblems of size $\frac{n}{2}$. But in general, we can divide the problem into b subproblems of size $\frac{n}{b}$, where a of those subproblems need to be solved.

This leads to a general recurrence for divide-and-conquer problems:

$$T(n) = aT(n/b) + f(n), \text{ where } f(n) \in \Theta(n^d), d \geq 0.$$

When we encounter a recurrence of this form, we can use the *master theorem* to determine the efficiency class of T:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Application of this theorem will often allow us to do a quick analysis of many divide-and-conquer algorithms without having to solve the recurrence in detail.

Empirical Analysis

Much of our work this semester will be a mathematical analysis of the algorithms we study. However, it is also often useful to perform an *empirical analysis* – counting operations in or timing an actual execution of an algorithm.

Let's see how we can perform a simple empirical analysis one of the algorithms we've considered: matrix-matrix multiplication.

See Example:
`/home/cs385/examples/MatMult`

Many factors make such an analysis difficult to perform with any degree of accuracy.

- System clock precision may be quite low. Some very fast operations may measure as 0.
- Subsequent runs of the same program may give different results.
  - Take the average? Take the minimum?
  - Modern operating systems are time shared – the "wall clock" time taken by your program may depend on other things happening in the system.
- As problem sizes vary, unexpected effects from cache and/or virtual memory may come into play.
- When considering algorithms whose performance depends on the input values as well as size, how do we choose data? Randomly? How best to achieve the true average case behavior?