Efficiency Analysis

We'll assume you are familiar with the concepts of "a problem," an "algorithm," and, to a limited extent, the analysis of algorithms, from CS 2050, say. If not, be sure to read some introductory material on algorithms.

However, we do need to note a few technical terms.

An *instance* of a problem is a particular collection of information that fits the pattern specified by the "problem." For example, an instance of the sorting problem is simply a finite sequence of items, each with a key, where the keys can be compared, in the sense of the `Comparable` interface in Java.

When we talk about how "efficient" an algorithm is, we always describe the time it takes to solve an instance of the problem "of size n." The size of an instance is, technically, the number of bits that it takes to represent the instance using a *sensible* representation.

> If we represent the instance is some really inefficient way, then we can make the algorithm look more efficient than it really is.
>
> Consider the problem of representing a single positive integer using a sequence of 0's and 1's. With our usual clever place value system, we can represent numbers of size 2^n using n bits, whereas cave people, making n marks to represent n, require 2^n bits to represent a number of size 2^n.

We won't worry too much about the input size issue, but consider the following efficiency analysis.

> Problem: given a positive integer n, either find factors $a > 1$ and $b > 1$ such that $n = ab$, or determine that no such factors exist (because n is prime).
>
> An algorithm to solve this problem: if n is divisible by 2, we're done. Otherwise, try 3, 5, 7, and so on, up to roughly the square root of n, and stop if any of them divide n. If not, n is prime.
>
> Analysis: this algorithm is (using CS 2050 notation to be made precise shortly) in $O(\sqrt{n})$, so the algorithm is very efficient and will run very quickly (after all, $O(\sqrt{n})$ is better than $O(n)$, and in CS 2050 we were usually totally happy with $O(n)$ algorithms).
>
> In fact, the security of RSA public key encryption (which we will study in this course) is entirely based on the idea that factoring a given n that is the product of two prime numbers is, as far as we know, impossibly time-consuming. What's going on? The problem is that the input size of the problem is not n, it is the number of bits needed to represent n. Just because we call some quantity "n" doesn't mean that it's the input size! If we instead say, given a positive integer N that takes n bits to represent, then the algorithm is indeed in $O(\sqrt{N})$, but N is in $O(2^n)$, so the algorithm is in $O(2^{n/2})$ which is in fact horribly time-consuming, as expected.
>
> So, we just have to be aware that we could delude ourselves by not measuring input size in a fair way.

When analyzing efficiency of an algorithm, we also need to be careful to note whether we are talking about the worst-case, the average case, or the best-case instance. These can be quite different for some algorithms.

Usually we will talk about the time efficiency of an algorithm, but sometimes the memory space that it requires is also analyzed.

Finally, even though we want to measure the time that an algorithm will take to execute on a given instance, we actually count the number of *representative operations* that are performed, with this measure giving us good insight into the performance of the algorithm if we choose what to count properly.

> For example, in our up-coming work analyzing sorting algorithms, we will count the number of comparisons that each of our algorithms requires. But, there are sorting algorithms (like bin sort) that do 0 comparisons, so comparison of keys is a terrible representative operation for those algorithms, and if we want to compare comparison-based algorithms with non-comparison-based algorithms, we'll have to be more thoughtful.

Asymptotic Analysis

We won't attempt to accurately count the number of representative operations that an algorithm takes on an instance. Instead, we will categorize algorithms according to their *asymptotic behavior.* This means that we only care about the general growth behavior, and only for large n. So, instead of painstakingly determining that some algorithm takes, say, $5n^2+11n+1006$ representative operations on an instance requiring n bits to represent it, we will be happy to say the algorithm "takes n squared time."

Now we want to formalize this idea. For all the functions we are going to be discussing, we will assume that the domain is integers $n \geq 1$, since n is the input size of some instance, and assume that the functions are always positive, since the functions measure the number of representative operations required to solve an instance of input size n.

Definition For functions f and g (satisfying the usual assumptions), we say that $f \in O(g)$ (read as "big oh of g") if and only if there is a positive integer N and a positive real number c such for all $n \geq N$, $f(n) \leq cg(n)$.

Example Let's use this definition to prove that $n^2 \in O(11n^2+107n+2007)$, and then to prove that $11n^2+107n+2007 \in O(n^2)$.

Showing the first one is easy—for all $n \geq 1$, $n^2 \leq 1 \cdot (11n^2+107n+2007)$, so with $N=1$ and $c=1$ we are done.

The second claim is harder, because in fact $11n^2+107n+2007$ grows more quickly than n^2. But, if we ignore the lower degree terms, it seems that it isn't growing faster than $11n^2$. So, we might be tempted to use $N=1$ and $c=11$, and try to prove that for all $n \geq 1$,

$$11n^2+107n+2007 \leq 11n^2.$$

Unfortunately, this isn't true no matter how far out we go. So, playing with N won't help—our idea of using $c = 11$ is just wrong.

But, if we take c a little bigger (in fact, any amount bigger than 11), we can prove the inequality for all n beyond some carefully chosen N. Let's demonstrate this with $c = 12$.

Example

Let's do the algebra to find precisely how large n has to be to guarantee that

$$11n^2 + 107n + 2007 \leq 12n^2.$$

First, we subtract terms from both sides to get the equivalent inequality

$$n^2 - 107n - 2007 \geq 0.$$

The quadratic formula tells us that the roots of this quadratic are

$$n = \frac{107 \pm \sqrt{107^2 - 4 \cdot 1 \cdot (-2007)}}{2 \cdot 1} = \frac{107 \pm \sqrt{19477}}{2} \approx \frac{107 \pm 139.56}{2}$$

which gives roots (approximately, of course) -16.28 and 123.28. So, we can take N to be any integer greater than 123.28, so we would use $N = 124$. Thus, we have proven the desired result because for all $n \geq 124$,

$$11n^2 + 107n + 2007 \leq 12n^2.$$

Doing the algebra is often easier, but we could also determine N by sampling to see how the two functions compare for larger and larger n (and understanding what the graph of a quadratic function looks like):

n	$11n^2 + 107n + 2007$	n^2
10	4177	1200
100	122007	120000
110	146877	145200
120	171440	172800
130	201817	202800

From these experiments (which if done by hand/calculator are probably more work than doing the quadratic formula), we could take $N = 130$, or try some more to narrow down the smallest N that works—but there's really no reason to do that!

The previous work is a little foolish, because we could take c to be any positive number we want. By picking bigger c, we might make our work easier. My favorite is to use $c = 11 + 107 + 2007 = 2125$, and note trivially that for all $n \geq 1$,

$$11n^2 + 107n + 2007 \leq 11n^2 + 107n^2 + 2007n^2 = (11 + 107 + 2007)n^2 = 2125n^2.$$

In fact, the closer we take c to 11, the farther out we have to go with N to make the desired inequality true from there on out.

Also, note that we could have experimented with $N = 100$, then $N = 200$ and that would work.

Note: the experimental approach is fine, but we do need to know (or hope) that we are taking N large enough that both functions have "settled down" to their asymptotic growth. If we happen to be experimenting with values of n that are smaller than some inflection points of the functions, we might come to incorrect conclusions about inequalities.

Definition For functions f and g (satisfying the usual assumptions), we say that $f \in \Omega(g)$ if and only if there is a positive integer N and a positive real number c such for all $n \geq N$, $f(n) \geq cg(n)$.

Definition For functions f and g, we say that $f \in \Theta(g)$ if there is a positive integer N and positive real numbers c_1 and c_2 such that for all $n \geq N$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Asymptotic Analysis of Polynomials is Trivial

The previous example is very typical. In fact, for any polynomial function, its efficiency category is $\Theta(n^d)$, where d is the degree of the polynomial (the highest power of n occurring in the polynomial).

At times in this course—and in life—we will encounter algorithms whose efficiency categories turn out to be non-polynomial functions, so we should be aware of how these functions compare, in general. Note that calculus provides a nice tool for this—if we can compute

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

then we can understand how f and g compare. In particular, if this limit is a constant, then f and g are in the same Θ category; if this limit is ∞, then f grows much more quickly than g, and if this limit is 0, then f grows much more slowly than g.

Of course, this is not helpful unless we can compute the limit (L'Hopital's rule might be handy), and sometimes it's more meaningful to directly prove comparisons.

Logarithms with Any Bases Have Same Growth Rate

It turns out that logarithm functions to all bases have the same growth rate (so we will usually use $\log_2()$ and write it as just "log"). To see this, let a and b be any real numbers greater than 1, and note, using basic logarithm properties, that

$$\log_b(x) = \log_b(a^{\log_a(x)}) = \log_a(x) \cdot \log_b(a) = \log_b(a) \cdot \log_a(x),$$

where $\log_b(a)$ is a positive constant.

This shows that $\log_b(x)$ is a constant, namely $\log_b(a)$, times $\log_a(x)$, which obviously shows that $\log_b(x) \in \Theta(\log_a(x)$, as desired.

⇒ Embedded Exercise 4

Figure out exactly why each claimed equality in this line above is true.

Some Sample Efficiency Analyses

We will develop whatever additional tools we need as required for each algorithm that we study, but right now we will analyze some familiar algorithms to begin developing the analysis tools.

We will use the sorting problem and two well-known algorithms that solve this problem, namely selection sort and merge sort.

We will use n as the number of items in the list being sorted.

> Note that n is not the actual input size. However, note that the actual input size is some fixed multiple of n, assuming that the keys are in some reasonable range of values, so the $\Theta()$ notation works just the same whether we use n or the actual input size.

We will use comparison of two keys of items in the array as the representative operation.

Selection Sort Algorithm

Recall the selection sort algorithm. On the first pass it will compare $n-1$ keys. On the second pass it will compare $n-2$ keys. Continuing in this way, we see that the algorithm, in every case, will do

$$(n-1) + (n-2) + \cdots + 1$$

comparisons. Recalling that $1+2+\cdots+n = \frac{n(n+1)}{2}$, we see that the selection sort algorithm is in $\Theta(n^2)$.

Another way to approach this analysis is to think of selection sort as recursive, which makes it easier to analyze. The recursive version says "to selection sort a given range of an array, say from index `first` to index `n-1`, find the smallest item in that range, swap it with the item in position `first`, and then recursively do selection sort on the range from `first+1` through `n-1`."

Then we can let $S(k)$ be the number of comparisons required to perform selection sort on k items, and note that

$$S(k) = k - 1 + S(k-1) = S(k-1) + k - 1,$$

because it takes $k-1$ comparisons to find the minimum of the k items being sorted, and it takes $S(k-1)$ comparisons to selection sort the rest of the items.
Now, this formula is true for any positive integer n, so we can apply it with $k = n$ and then $k = n-1$, obtaining

$$S(n) = S(n-1) + n - 1 = [S(n-2) + (n-2)] + n - 1 = S(n-2) + (n-1) + (n-2).$$

Then we can apply it with $k = n-2$, and so on, obtaining (after noting the obvious pattern),

$$S(n) = S(n-2) + (n-1) + (n-2)$$

$$= [S(n-3) + (n-3)] + (n-1) + (n-2) = S(n-3) + (n-3) + (n-2) + (n-1)$$

$$\vdots$$

$$= S(n-n) + (n-n) + (n-(n-1)) + \cdots + (n-3) + (n-2) + (n-1) =$$

$$S(0) + 0 + 1 + \cdots + (n-2) + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Note that this reasoning could (and maybe should) be rigorously proven by math induction, but we generally won't bother.

⇒ **Embedded Exercise 5**

Make sure that you understand every step in the previous argument.

Merge Sort Algorithm

Recall the merge sort algorithm. To analyze it, we use the same substitution technique we just used for selection sort. To do this, we let $T(n)$ be the number of comparisons required to perform merge sort on an array of n items.

To simplify analysis, we will only concern ourselves with the elegant case that n is a power of 2, say $n = 2^m$.

Merge sort works by splitting the list into two halves and recursively merge sorting each half, followed by merging the two sorted halves. Since the merge operation clearly takes a comparison for each number that is moved into temporary storage, we see that for any k (which is a power of 2),

$$T(k) = T(k/2) + T(k/2)k = 2T(k/2) + k.$$

Or, applying this with $k = 2^m$,

$$T(2^m) = 2T(2^{m-1}) + 2^m.$$

Then if we apply the recurrence to the $T(2^{m-1})$, using $k = 2^{m-1}$, we obtain

$$\begin{aligned} T(2^m) &= 2[2T(2^{m-2}) + 2^{m-1}] + 2^m \\ &= 2^2T(2^{m-2}) + 2^m + 2^m. \end{aligned}$$

⇒ **Embedded Exercise 6**

Continue doing the same substitution until the pattern becomes clear and you can express $T(2^m)$ in a non-recursive way and hence determine the $\Theta()$ category for the number of comparisons required to merge sort a list of $n = 2^m$ items. We must express the final result in terms of n.

Two Sums You Must Know

The previous two examples ended up using two famous sums that you absolutely have to be comfortable with, namely

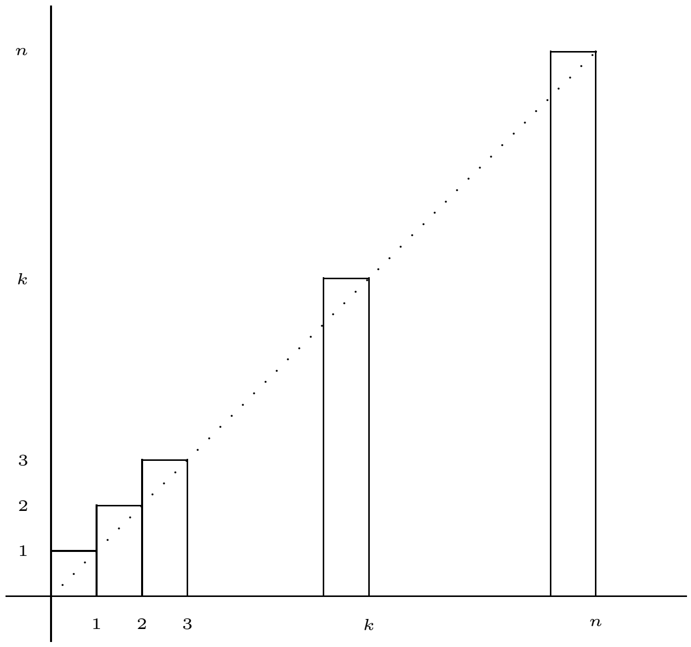
$$1 + 2 + 3 + \cdots + n$$

and

$$1 + r + r^2 + \cdots + r^k$$

⇒ **Embedded Exercise 7**

a. Prove the fact that $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ geometrically by figuring out the area of each rectangle in this diagram, and then figuring out the sum of the areas of all the rectangles.



b. Prove that

$$1 + r + r^2 + \cdots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

as follows. Let

$$S = 1 + r + r^2 + \cdots + r^k.$$

Then

$$rS = r + r^2 + \cdots + r^k + r^{k+1}.$$

Carefully compute $rS - S$ (noting that most the terms cancel), set that result equal to $S(r-1)$, and solve for S.

For Asymptotic Analysis, You Don't Need to Know any Sums, Really

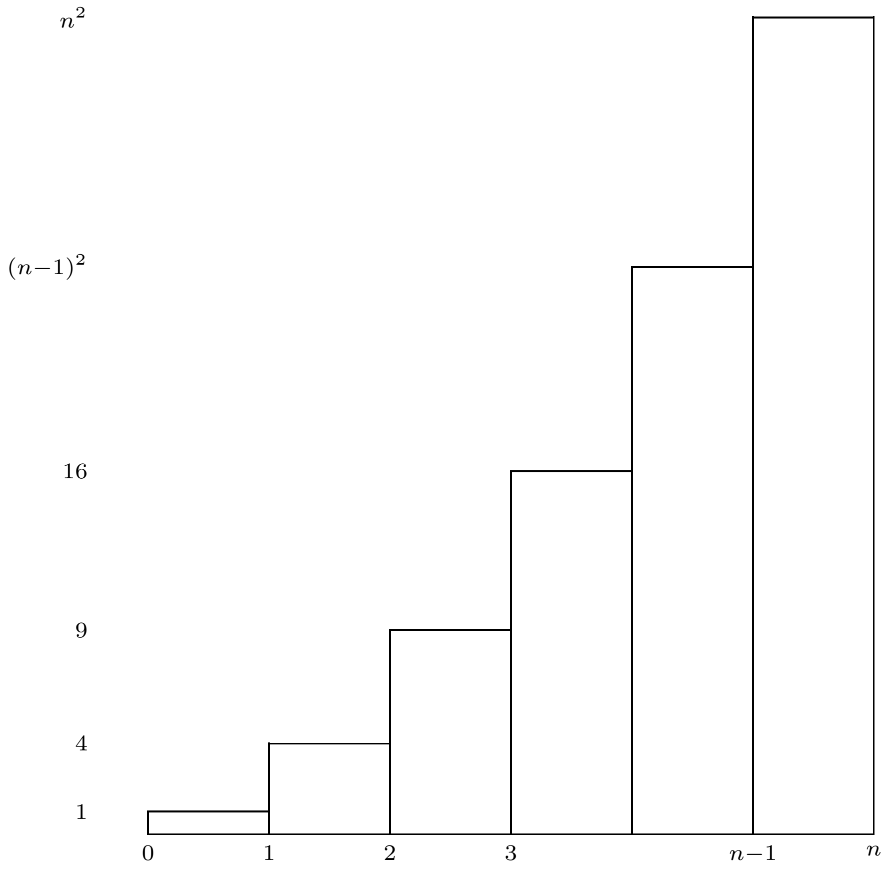
Suppose you figured out that some algorithm yields the pattern

$$A(n) = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

and you want to say what Θ category this function belongs to. You could go look for a formula for this sum (there is in fact a formula for every sum of the form $\Sigma_{k=1}^{n} k^d$ for any positive integer d), but you wouldn't have to. This is part of the appeal of asymptotic analysis—we just want to generally know how fast a function grows, not exactly what its value is.

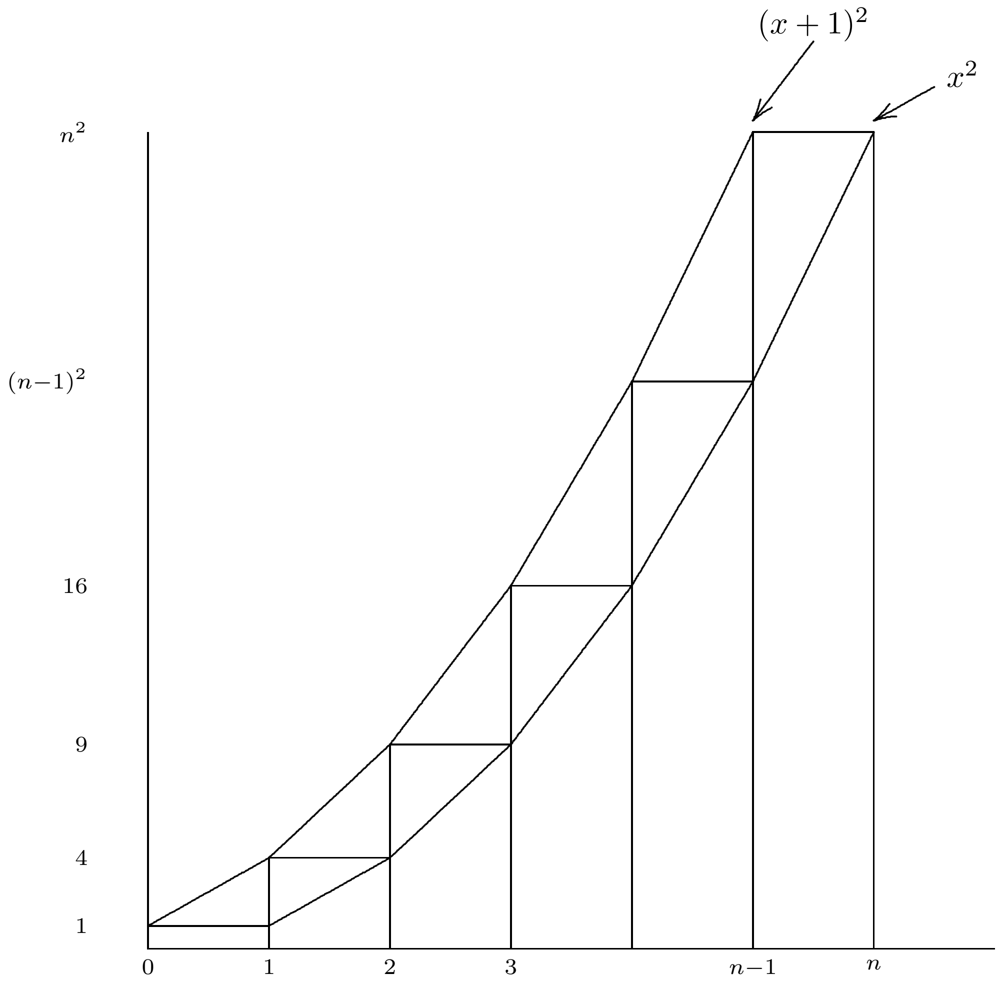
The technique, which we actually used earlier to compute $1 + 2 + \cdots + n$, is to find definite integrals that bound the function, represented as a sum of rectangles. If we can compute the definite integrals, then those results bound the target function.

To apply this powerful technique, we first draw rectangles whose areas add up to $A(n)$, like so (where to fit the page with accurate scaling we use $n = 6$):



Then we figure the total area of the rectangles, which is $1+4+9+\cdots+(n-1)^2+n^2 = A(n)$ since each rectangle has width 1.

Then we can sketch in two functions—one that bounds the area of the rectangles below, and one that bounds it above:



After some thought we can figure out that the functions are as given in this diagram—the one that passes through the right endpoints is easy, because it is the same as the heights of the rectangles, and the other is just that one shifted somewhat.

Now, we know that

$$\int_1^n x^2 dx \leq A(n),$$

and

$$\int_0^{n-1} (x+1)^2 dx \geq A(n),$$

so we can determine the Θ category of $A(n)$ by actually computing these two definite integrals and finding out that they are both in $\Theta(n^3)$.

$\Rightarrow$ **Embedded Exercise 8**

Make sure that you understand the various claims made in the previous discussion, and then actually do these two integrals carefully and determine the Θ category for both.

Finding the Growth Rate of the Fibonacci Sequence

Note: this material is included in case you might be curious, but no further work on this will be done.

Sometimes efficiency analysis can be more difficult. As an example of this, and because we will want the result later when we care about the efficiency category of the Euclidean algorithm, let's analyze the growth rate of the Fibonacci sequence.

Consider the famous Fibonacci sequence defined by

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

⇒ **Embedded Exercise 9**

Compute the first ten or so values of F.

We want to determine the Θ category for F. Recall from your discrete math course (or just believe it) that functions like F that fit a recurrence relation like this:

$$F(n) - F(n-1) - F(n-2) = 0$$

have solutions of the form

$$F(n) = ac^n + bd^n$$

where a, b, c, and d are cunningly chosen real numbers.

To find the numbers, we can first note that if $c^2 - c - 1 = 0$ and $d^2 - d - 1 = 0$, then $F(n) = ac^n + bd^n$ satisfies the recurrence relation for any a and b. Note that this goes in the direction "if these guys are nice then the recurrence relation is satisfied" but says nothing about whether a solution to the recurrence relation *has* to be of this form.

We can find c and d as distinct numbers by solving the quadratic $x^2 - x - 1 = 0$, finding that $c = \frac{1+\sqrt{5}}{2} \approx 1.618$ and $d = \frac{1-\sqrt{5}}{2} \approx -0.618$.

Finally, we can use the initial conditions $F(0) = 1$ and $F(1) = 1$ to determine a and b. Note that different initial conditions lead to different values for a and b.

If we do all this correctly, we find that $a = \frac{c}{\sqrt{5}}$, $b = -\frac{d}{\sqrt{5}}$, and

$$F(n) = ac^n + bd^n$$

where c and d are as above. The d^n term goes to zero very quickly, so we have shown that $F(n) \in \Theta(\left(\frac{1+\sqrt{5}}{2}\right)^n)$. Thus, this function grows more slowly than 2^n, and more quickly than $\left(\frac{3}{2}\right)^n$, to put things in a way that only involves integers. To use our formal notation, we have shown that $F \in O(2^n)$ and $F \in \Omega(\left(\frac{3}{2}\right)^n)$.

⇒ Project 5 [routine] Working with the Definitions

This project is intended to give you some simple, concrete experience with the formal definitions of O, Ω, and Θ.

Let $f(n) = n^3$. For each of the functions g given below, formally prove, directly from the definition, that $g \in \Theta(f)$. Each of your 6 separate proofs must clearly state N and c as in the definitions of $O()$ and $\Omega()$, and must clearly explain the algebra or experimentation supporting your inequalities.

a. $g(n) = 1000n^3$

b. $g(n) = n^3 + 1000n^2$

c. $g(n) = n^3 - 1000n^2$ (assume $n \geq 1000$ for this to be a legitimate function for our purposes)

⇒ Project 6 [routine] Solving a Reduce and Conquer Recurrence Relation

Recall the selection sort algorithm. If we let $S(n)$ be the number of comparisons required to sort n items using selection sort, then it is easy to see that for any $k > 1$,

$$S(k) = k - 1 + S(k-1) = S(k-1) + k - 1,$$

because it takes $k - 1$ comparisons to find the minimum of the k items being sorted, and it takes $S(k-1)$ comparisons to perform selection sort on the rest of the items.

You job on this problem is to solve this recurrence relation (also using the obvious fact that $S(1) = 0$). In other words, you need to come up with a non-recursive formula for $S(n)$.

With $k = n$, you have

$$S(n) = S(n-1) + n - 1 = [S(n-2) + (n-2)] + n - 1 = S(n-2) + (n-1) + (n-2).$$

Continue in this way—substitute the recursive formula for $k = n - 2$, and continue until you see the pattern, and can express $S(n)$ as a sum of terms with no recursive call involved. Then recall some basic math (the sum of $1 + 2 + \cdots + n - 1 + n = ?$) to express the sum as a single algebraic expression.

⇒ Project 7 [routine] Solving a Divide and Conquer Recurrence Relation

In our upcoming work on algorithms designed using the *divide and conquer* technique, we will see that the number of representative operations $T(n)$ for an instance of size n will be defined in terms of recursive calls to the same function T. For example, here are some efficiency functions for algorithms you have already seen, or will see:

$T(n) = T(n/2) + 1$ (binary search)

$T(n) = 2T(n/2) + n$ (merge sort)

$T(n) = 3T(n/2) + n$ (Karatsuba's algorithm)

$T(n) = 7T(n/2)$ (Strassen's algorithm—multiplications)

$T(n) = 7T(n/2) + 18\left(\frac{n}{2}\right)^2$ (Strassen's algorithm—additions)

In view of these recurrence relations, and with other possibilities in mind, we want to solve, by the substitution technique, any recurrence relation of the form

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d,$$

where, to keep things nice, we will only consider n that are powers of b (typically $b = 2$, but it is easy to consider any $b > 1$), i.e., $n = b^m$, and a, c, and d are constants.

Your task is to solve this recurrence relation (assume that $T(1)$ is a known number). To get you started:

$$\begin{aligned} T(n) = T(b^m) = aT(b^{m-1}) + c(b^m)^d = aT(b^{m-1}) + cb^{md} &= aT(b^{m-1}) + c(b^d)^m \\ &= aT(b^{m-1}) + cp^m \end{aligned}$$

where we use $p = b^d$ for convenience.

Now, we substitute this recurrence in for $T(b^{m-1})$, obtaining

$$T(b^m) = a[aT(b^{m-2}) + cp^{m-1}] + cp^m = a^2T(b^{m-2}) + c(a^1p^{m-1} + a^0p^m).$$

Continue in this way for a few more terms, until you see the pattern. Then extrapolate (you could do math induction to proof your conclusion, but we won't bother) to obtain a formula for $T(b^m)$ that doesn't include any calls to T (since $T(1)$ is a known constant). This formula will include a sum of terms.

Now, solve each of the specific recurrence relations given by plugging in the values for a, b, c, and d and adding up the sum in each case.

Once you have done the algebra, state carefully, in terms of n (not m) the Θ efficiency category for the function T.

⇒ Project 8 [optional] Average Case Analysis of Quicksort

To show that occasionally we are interested in the *average case* behavior for an algorithm, let's look at the average case behavior of quicksort.

This project requires you to nicely write up, with all details explained, the entire proof that is sketched below.

Quicksort is a famous divide-and-conquer algorithm. It is well-known that for any particular strategy for picking the pivot item in the partition algorithm—say picking the first item in the list to be partitioned—quicksort can have worst-case behavior in $\Omega(n^2)$. So, why does the algorithm continue to find favor? We will present a proof that quicksort has $\Theta(n \log n)$ *average-case* performance.

Let $A(n)$ be the average time for quicksort to sort n items and find a recurrence relation for this function, which we will then solve.

We assume that given n items to partition, all n are equally likely. So,

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1.$$

This is true because each of the n possible choices of pivot item location is equally likely, each with a probability of $1/n$, and the partition time is $n-1$ independent of the location of the pivot time, on average, and $A(p-1)$ and $A(n-p)$ are the average times needed to do the two recursive calls to quicksort on the two sublists produced by the partition.

⇒ Show that

$$\sum_{p=1}^{n}[A(p-1) + A(n-p)] = 2\sum_{p=1}^{n} A(p-1)$$

So,

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1.$$

So, we have the relationship

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1).$$

Now we do a clever technique, using the same relationship with a different index, substituting $n-1$ for n to obtain

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2).$$

Then we subtract these two equations, yielding

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + n(n-1) - (n-1)(n-2) = 2A(n-1) + 2(n-1).$$

⇒ Verify this algebra.

Now, tidy things up to yield

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

This might have all been pointless, except we sort of lucked out and the two $A()$ terms look the same, but with the argument shifted, so we can define a new function by

$$B(n) = \frac{A(n)}{n+1},$$

and see that the previous equation gives a recurrence relation for $B(n)$, namely

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)}.$$

⇒ Now we need to do our usual substitution technique to find a pattern for $B(n)$, and then bound the resulting sum (which is neither a geometric series nor an arithmetic series, which are pretty much the only ones we know how to add up exactly) by two integrals, both of which integrate to some multiple of the natural logarithm of n. Thus, $B(n) \in \Theta(\log n)$, so $A(n) \in \Theta(n \log n)$, by the definition of $B(n)$.

$\Rightarrow$ As a whole group, wrap up Project 7. Make sure every group got correct algebra, then apply to the divide-and-conquer examples given and produce finally Θ result in terms of n for each.