Chapter 35

Big-O notation

In a previous section about making a faster program I wrote that none of the obvious tricks worked very well. We can mess around rearranging `if`s and equations and, if we turn our program into an incomprehensible bug-ridden mess and are very lucky, maybe we get a x2 speed-up.

Profiling is just more of that same thing. All it does is help you use those little tricks, like telling you which functions get used the most. Or, after you rearrange some `if`'s it will tell you the function is *slower* so you need to keep trying. In a game, it will often tell you the program is plenty fast, but the graphics card is being swamped. You just need to use pictures with a smaller pixel-count.

Profilers are great, especially for code written by a lot of people which you suspect is sloppily assembled. If you're going to do that stuff, go ahead and use one. But it's still checking the couch for spare change compare to the real speed-up trick.

The one thing that gives a huge speed-up is avoiding extra loops. If you have a loop that runs 1,000 times, and you realize you can do the same thing without one, that's about a 1,000 times speed-up.

Avoiding unnecessary loops is the major trick for worrying about speed.

Nothing in this chapter will let you do things you couldn't do before, or make your program look nicer or anything else useful. It just does one thing, which is make it faster. But if you're lucky, it will make it *a lot* faster.

The first section explains that idea more. The second part has examples of using it to classify common array loops; then examples of removing loops by using a better plan. Then a little about how the loops are different for a linked-list.

The section after that introduces the formal way we write big-O notation. It's useful to know because it's the way computer manuals list speeds. Then

there's more examples, data structures and the last section is a summary of how we use this thinking in general program writing.

35.1 Loop counting logic

The math and motivation are long and boring, so I'm going to skip ahead to the end results and what we do about it. I'll go over the explanation in a section at the end, and hopefully the examples will help. Just keep in mind there will be a few places where I'm just saying something is true without really showing why.

To help speed up programs, we classify every function by the worst nested loop it has. We don't worry about how many times the loops run, or how many lines are in them. We list them as only: not a loop, single loop, nested loop, triple loop and so on.

Just to make sure, here's a sample triple loop, using 3 arrays:

```
for(int i=0;i<A.Length;i++)
  for(int j=0;j<B.Length;j++) {
    count1++;
    for(int k=0;k<C.Length;k++)
      count2++;
    }
```

If all the arrays are about 1,000 long, `count1` will be a million and `count2` will be a billion. Each nesting is 1,000 times more steps.

That's why we simplify so much. In general, any single loop is much worse than any non-loop, any nested loop is much worse than any single loop, and so on.

After we label a function with the nested level, we try to reduce it. We don't waste time trying to make the loop run fewer times, or do less work each time. All we do is try to think of a plan that uses less nesting. That's where we get the big x100 speed-ups.

If we have a single loop, we try to think of a way to not use a loop. If we have a nested loop, we try to think of a way to do the same thing with only a single loop. And so on.

Again, I'm not really explaining the math or the thinking, but here are some comments why this plan sort of makes sense:

- This system says that non-loops are the simplest thing and all count the same, as one step. That's pretty much true. Compared to a big loop, the most complicated math and `if`s is insignificant. And we can't get a good x100 speed up for them, since they don't have any loops to remove.
- If you speed up a single loop by running it fewer times, or having less lines inside, this system says it's still a single loop, so those speed-ups

didn't matter. That's also pretty much true. Those are just x2 speed ups. Removing the loop would get you x100.

- A nested loop might run quickly on something small, and not be a problem at all. But sizes tend to grow and it will be. For example, a nested loop on 12 cats might be only 144 steps. But as the pet store grows we'll have 100 cats and the loop will take 10,000 steps.
- After we've eliminated extra loops, the classifications give a good estimate of speed. If a function we want is labelled as a nested loop, yikes! If we plan to put a loop around it we may want to think of a better way.
- A function with two loops in a row doesn't seem to fit in our system, but it counts as a single loop. It's just twice as slow as one loop, which doesn't matter. We have to eliminate both to make it a non-loop (getting rid of just one is like a useless x2 speed-up.)
- Sometimes we have fixed, tiny loops, and they obviously don't count. For example, a 9 step nested loop to set up a Tic-Tac-Tow board.

  It's usually pretty obvious. We have either very small loops that will never grow, or loops running 100+ times which will eventually get larger.

35.2 Integer array loops

The standard loops (or non-loops) on int arrays make good examples of how to apply the classifications.

Finding an item at a certain position in a list is a one step non-loop. We knew that, but I wanted to point out we don't always have to loop to do things with an array.

The first item is one step, `A[0]`, but finding the middle and last are three: `A[A.Length/2]` and `A[A.Length-1]`. We don't even care. It falls under the "basically 1 step" non-loop rule.

Counting how many times something occurs in an array is a single loop:

```
int howManyOf(int[] A, int countMe) {
  int count=0;
  for(int i=0;i<A.Length;i++) if(A[i]==countMe) count++;
  return count;
}
```

A funny thing is it takes longer depending how many there are (`count++;` takes a step.) We can ignore whether the inside runs 1 or 2 steps and just count it as a loop.

Checking whether something is in an array is even more variable. It might be the first item, finding it halfway is average, and if it's not there we always run the full loop:

```
bool contains(int[] A, int findMe) {
  bool found=false;
  for(int i=0;i<A.Length; i++) if(A[i]==findMe) { found=true; break; }
  return found;
}
```

This is really about the best, average and worst time a loop takes. Usually those are about the same – within half or double of each other. In this case the minimum of 1 or 2 steps is obviously very rare. The average is obviously a loop.

A way to look at it is the `break;` is like a x2 speed up trick. It's better than nothing, but it's still a loop.

Some loops average only 1 or 2 steps. They almost never run the full time. We'd probably count them as non-loops (the math can be tricky,) but those are rare and you'll know them when you see them.

Finding the smallest item is also variable speed But it still easily counts as a standard big-O loop:

```
int indexSmall=0;
for(int i=1;i<A.Length;i++)
  if(A[i]<A[indexSmall]) indexSmall=i;
```

I like how we starting the loop at 1 not so much for a tiny speed-up, but more since it seems weird to compare `A[0]` with itself.

Checking if a loop has duplicates is our first nested loop. The standard way is for the inner loop to only check items *after* us, and to quit early if we find a match:

```
bool hasDuplicates(int[] A) {
  for(int i=0;i<A.Length-1;i++)
    // compare to everything after it:
    for(int j=i+1;j<A.Length;j++) if(A[i]==A[j]) return true;
  return false;
}
```

This is another example where we ignore small tricks. This runs only 1/2 a million steps if there are no duplicates, and half that (on average) if there is one. But that's still about the same speed as a regular nested loop, and much, much slower than a single loop would be.

Finding the most common item is another nested loop, which shows off how our categories might help. This is a modified "largest item" loop, except it compares how many copies:

```
int mostCommonItem(int[] A) {
  int mostNum=-999;
  int mostCount=-1; // anyone can beat this
  for(int i=0;i<A.Length;i++) {
    int cur=A[i];
    int count=howManyOf(A, cur);
    if(count>mostCount) { mostNum=cur; mostCount=count; }
  }
}
```

We already labelled `howManyOf` as a loop. So this whole thing is a nested loop. We probably would have seen this on our own. The big-O idea of thinking only of loops just makes it a little easier.

Inserting and removing items from a list is funny, since they change the size. A common trick is having a too-big array and an extra `size` variable.

With that, adding and removing from the end is 1 step: insert is `A[size]=newVal; size++;` and remove is `size--;`. Technically insert is 3 steps, but the important thing is they're both not loops.

Removing from the middle is a loop. Everything past it slides back to fill the hole. That's probably about 1/2 the array on average (does our program like to delete nearer the start or the end?) It counts as a loop:

```
// remove from middle:
// slide forward everything past removed item, to fill the hole:
for(int i=remIndex+1;i<size;i++) A[i-1]=A[i];
size--;
```

Sorted Arrays

Suppose an array is sorted. Sometimes it's not too hard to keep it sorted as we go. Or maybe it won't take too long to sort it when we need to. If it's sorted, there are a few things we can do faster:

Finding the smallest and largest now take only 1 step (first item, last item.) This seems like cheating, since sorting obviously takes longer than a loop. But if it was already sorted, 1 step is pretty cool.

Checking for duplicates in a sorted list is only a single loop: check whether each item is the same as the one next to it. This loop starts at 1 and compares to the item before it:

```
bool hasDuplicatesSorted(int[] A) {
  for(int i=1;i<A.Length;i++) if(A[i]==A[i-1]) return true;
  return false;
}
```

We removed a loop, sort of. Duplicates on a size 1,000 list went from 250,000 steps to about 750.

If you remember, finding an item in a sorted list can be done with a binary search, which is an odd cut-in-half loop. You find which half of the array your item should be in (which you can do since it's sorted,) then find which half of that half it should be in, and so on.

The important thing is it's much faster than a regular loop, but it's still sort of a loop. For 1,000 items it takes 10 steps. It takes 20 steps for a million length list. We'll make a new category for this, and call it a "cut-in-half" loop.

The end result is searching for an item went from about 500 steps to 10 steps. We almost removed a loop – 50 times faster is pretty impressive.

Just for fun, here binary search, which you can skip. I wrote a binary search in the recursion section, but we can write it as a loop. This picks a middle, figures out the half we should look in, and repeats until we're down to 1 or 0 items in our half:

```
bool isInSorted(int[] A, int findMe) {
  int start=0, end=A.Length-1;
  while(start<end) { // at least 2 things in it
    int mid=(start+end)/2;
    if(findMe<=A[mid]) end=mid;
    else start=mid+1;
  }
  if(start>end) return false; // 0 items
  return A[start]==findMe; // 1 item. Is it our number?
}
```

Finding the most common item is down from a nested loop to a single loop. Identical items are in a row, so we can count as part of the only loop (there's code for this later, but the idea should be clear enough.)

Sorting

Sometimes the array just happens to be sorted. Often it's almost sorted and a few easy changes can make it completely sorted. Even if it's not, it's often faster to sort it, then check for whatever you wanted.

The easy sorts are nested loops: bubble sort, selection sort and insertion sort. But sorting is pretty important and over the decades we've figured out some faster, but complicated ways of doing it. The built-in sort, `Array.Sort(A);`, uses one of those.

Fast sorts are a nested loop where one is just a cut-in-half loop. That means sorts are just a little bit worse than a single loop. For a length 1,000 array, the

cut-in-half loop runs 10 times, for only 10,000 steps total (much better than hundreds of thousands for a nested sort.)

If you want to look them up, the fast sorts are named merge sort, heap sort and quick sort. They all work differently, but all use a regular loop nested with a cut-in-half loop.

This means for anything that takes a real nested loop – duplicates or finding the most common – we can usually do it a few hundred times faster by sorting first, than using a single loop.

Suppose we want to check whether two arrays are the same. If would be a nested loop. But we can go faster by sorting both, then using a single loop to compare items side-by-side.

35.2.1 Fun with bad loops

C# has a class `List` which is really just a wrapper around an array. For example, `B.RemoveAt(50);` gets rid of `B[50]` by running the slide-down loop. It's made for when you want an easy-to-use array and don't care about speed.

Here's a quick demo using it:

```
List<int> B = new List<int>{4,12,8,29}; // C# allows this shortcut
for(int i=50;i<100;i+=5;) B.Add(i); // adds 50, 55, ... to the end. B will grow
int n=B[3]; // normal indexing. n is 65
int i1=B.IndexOf(8); // search position of 8 (this is a loop)
B.RemoveAt(7); // slide-down loop
B.Remove(29); // search loop + slide down. Counts as 1 loop
B.Insert(7, 785); // insert 7 at position 785. Slide-back loop
```

If you don't think about which of these are loops, you can write some slow programs. Suppose we want to get rid of all negative numbers. We can use a back-to-front loop, deleting as we see them (going backwards makes it easier not to skip over a number when we delete):

```
for(int i=B.Count-1; i>=0; i--)
  if(B[i]<0) B.RemoveAt(i);
```

This looks pretty short, but `RemoveAt` is a loop, so this is a nested loop. Yikes.

It turns out we can write a multiple remove using a single loop. There's a built-in `RemoveAll` that works that way, but it requires you know how to send a function: `B.RemoveAll(n=>n<0)` removes negative numbers from `B`.

Just for fun, this is a single loop that creates a new array with all the negative numbers removed. The last loop is a standard "copy only some things," using a floating index for the new array:

```
// count how long the new array should be:
int len=0;
for(int i=0;i<A.Length;i++) if(A[i]>=0) len++;
int[] Temp = new int[len];
// copy positive#'s into new array:
int i2=0; // next open spot in Temp
for(int i=0;i<A.Length;i++)
  if(A[i]>=0) { Temp[i2]=A[i]; i2++; }
  // else leave this out by not copying it
A=Temp;
```

Another accidental bad loop: suppose we're trying to make a list going from 99 down to 1. A clever plan is to count forwards, from 1 to 99, but add each item to the *front* of the list:

```
List<int> B = new List<int>(); // size 0
for(int i=1; i<=99; i++) B.Insert(0,i); // add to front (position 0)
```

But we know inserting to the front is a loop, so this is an unnecessary nested loop.

35.3 Linked Lists

If you haven't seen linked lists, the important part is that each item is connected by pointers to the ones before and after. This means that if we have an item, we can remove it by just changing a few pointers. We can also easily insert to the front, back, or the middle.

But this also means we can't jump to `A[50]` in one step. We need a loop from the front following the next-item pointer 49 times. That also means we can't do a binary search – we can't just jump to the middle.

The example where we make 99 to 1 by adding 1 to 99 to the front works fine with a linked list. This is a single loop:

```
LinkedList<int> N = new LinkedList<int>(); // empty list
for(int i=1;i<=99;i++) {
  N.AddFirst(i); // only 1 step
}
```

The main reason we use linked lists is for times when you'll loop through the whole list, and delete items where-ever you find them.

For example, assume we have a linked list `K` and occasionally add new killer robots to it. This processes everything in the list, removing dead ones. It's a little long. The main idea is that "remove current item" is one step:

```
// loop variable is a pointer. Start it at the first one:
```

```
LinkedListNode<KillerRobot> ptr = K.First;
  while(ptr!=null) {
    // in case we have to delete this one, get pointer to the next robot now:
    LinkedListNode<KillerRobot> ptrNext=ptr.Next;
    KillerRobot kr = ptr.Value; // current robot
    // process kr somehow
    if(kr.health<0) {
      K.Remove(ptr); // takes only a 1 step, instead of a loop for an array
    }
    ptr=ptrNext; // will be null if last item
  }
```

By switching from an array to a linked list, we turned a nested loop into a single loop. Of course, we can only use that trick if this is the only thing we do. If some other function searches or wants to jump around we'll have to think more.

35.4 Formal math

Instead of talking about loops, we really measure speeds using formulas. They're shorter and more accurate. For example, a nested loop over 1,000 items takes 1,000 x 1,000 steps, which is the size squared. We just write that as $O(n^2)$.

The rules are:

- A capital O and parens goes around the whole thing. This is why it's called big-O notation.
- If there's no loop just write $O(1)$, which stands for "basically one step."
- Pick a variable to stand for the size of the input, usually n. This is often the size of the array.
- Write an equation using n with no constants and only the highest term. This is pretty much what we've been doing before.

The numbers of times you can cut something in half is the *log* (really the log base-2, but they're all the same except for multiplication.) The times we already know work out as:

$O(1)$ - not a loop
$O(log(n))$ - cut in half loop
$O(n)$ - single loop
$O(n * log(n))$ - good sorting loop (cut-in-half around a normal loop)
$O(n^2)$ - nested loop
$O(n^3)$ - triple nested loop

If you read technical stuff, they'll tell you inserting to an array is $O(n)$, which means it takes about n steps where n is the size of the array.

Formally we look at these as asymptotically. That means can can graph lots of things as n increases, step way back and we'll see these categories. If you graph $3n^2$ and $n^2 + n$ they'll run very close to just n^2 (when n gets large enough.) That's the formal math part of why we simplify to no constants and the largest term.

There is a formal math definition of big-O, which someone had to figure out and prove it made sense, but you can safely skip it.

There are an infinite about of big-O categories, for example $O(\sqrt{n})$ or $O(n^{1.9})$. But the ones I listed are the ones that come up in real programs. For example, you could write a loop that takes the square root of the size and spins that many times, and it would be $O(\sqrt{n})$. But there's almost no real problems you'd solve that way.

Cut-in-half loops are the only common funny ones, which are $O(log(n))$. If you're wondering, a cut-in-thirds loop is also $O(log(n))$. It's about 30% fewer loops, but more work inside, so it evens out.

Sometimes you use two variables. For example, a loop that compares every item in one array to every item in another would technically take $O(m \cdot n)$, where those are the two array sizes. But they're usually pretty close so calling it $O(n^2)$ is fine.

35.5 More tricks and exceptions

Obviously, not all loops are problems, and not all problems are loops. We're really looking for things that more and more time as the input grows. A big array loop is just the most common way that happens.

Some notes:

Small, fixed-sized loops don't count. For example, suppose you're on a grid and want to check the 5x5 area near you. A nice way to do it is with this nested loop:

```
// get sides of 5x5 area around me (me -2 and +2) not off edge:
int x1 = Mathf.Max(myX-2,0), x2=Mathf.Min(myX+2,width-1);
int y1 = Mathf.Max(myY-2,0), y2=Mathf.Min(myY+2,height-1);
for(int x=x1;x<=x2;x++)
  for(int y=y1;y<=y2;y++) {
    if(x==myX && y=myY) continue; // skip space I'm on
    // check Grid[x][y] ...
  }
```

This runs at most 25 times. If the grid grows to 5,000 by 5,000 – this still runs 25 times. Since we're really trying to estimate the time it takes, we call this $O(1)$.

In Unity3D, `GetComponent` and `transform.Find` are also often tiny fixed-sized loops that should count as $O(1)$. GetComponent loops through all of your components until it finds the one you wanted. But I've never had a gameObject with more than six components. Realistically, it's 6 steps and counts as $O(1)$.

Likewise I often use `transform.Find` when I know I have two children and will never have any more (for example, a label with children: Text and Background.) It loops through them, but it's 2 steps, tops, so we'd call it $O(1)$.

`GameObject.Find` is more like a real $O(n)$ loop, since you probably have hundreds of game objects, and will have more in the future.

This is an example of linked list vs array. They decided GameObjects are in a linked list to make removing them faster. That makes finding one slower, but it's easy to program around that loop by just saving links to the ones you need.

For more fun, some people like to use empty gameObjects as folders, so `transform.Find` could count as $O(n)$. For example, we put a few hundred pickups inside of an empty named `pickupHolder`. Now it would be reasonable to say `transform.Find("healthPack5")` is $O(n)$.

To see why that matters, suppose we do something with every health pickup, using a loop to get health pack 0, 1 and so on:

```
for(int i=0;i<maxPacks;i++) {
  Transform tt = pickupHolder.Find("healthPack"+i); // <-big loop
  // do something with tt
}
```

If we're thinking in big-O terms, we might spot this as a nested loop, running about 100 times slower than it should. We could get a massive speedup rewriting as a single loop that goes through all children, skipping the non-health packs.

Count the work, not the loops. Sometimes a nested loop just goes through the array once, so is really $O(n)$. For example this finds the longest run of numbers in a sorted array (if you don't want to read it, the important thing it's a natural way to solve this, and looks like a nested loop):

```
int maxRunLen=-1, maxRunVal=-999;
for(int i=0;i<A.Length;) { // will increase i inside the loop
  int runLen=1; int val=A[i]; // set-up to count this number
  i++;
  // count the duplicates, check for a new winner:
  while(i<A.Length && A[i]==val) { i++; runLen++; }
```

```
  if(runLen>maxRunLen) { maxRunLen=runLen; maxRunVal=val; }
}
```

If you trace it, this goes through the array just one time. Both loops push `i` ahead. If the array has 1,000 items, this takes 1,000 steps. So it's really $O(n)$.

Likewise, it's possible to write a single loop that runs like nested loops (use `if`s to move `i` and `j`.) And we're already seen the cut-in-half loop.

Non-array loops count. These aren't as common, but sometimes you just loop over numbers. If you check every angle from 0 to 180, going by 0.1, that's about 2,000 steps. A nested loop might check a different angle on 0-90 by 0.1's. It seems fair to call that $O(n^2)$, and to try to think of a less loop-using way.

Sometimes you have to pick `n`. If a loop counts up to a number, `n` is the number. But if a loop checks each digit of a number, `n` is just the number of digits, which isn't that large (but a double or triple-nested loop can still get big, fast.)

If you have a 50x50 grid you might say n is 50, or you might think of n as 2,500 (which is the number of actual spaces, but is also 50 squared.) If the grid is more rectangular, like 10x1000, n = the number of squares probably makes more sense.

If seems funny that we can just pick it, but we're only using it for comparison. It a "pick one and stick with it" situation.

Recursion counts, and is tricky. A loop is the most common way to spin 1,000 times. But if you run through an array using recursive calls, time is time. For example, this recursive loop-faking function is $O(n)$:

```
bool allPositive(int[] A, int startIndex=0) {
  if(startIndex>=A.Length) return true; // made it to the end w/o quiting
  if(A[startIndex]<=0) return false; // found a negative
  return allPositive(A, startIndex+1); // keep looking
}
```

Recursive functions that call themselves twice or more can be tricky. Some blast out of control with $O(2^n)$ – exponential time (which is as bad as it sounds – worse than the most nested loop.) But some work out to just $O(n)$, like flood-fill only sees each square 4 times.

The recursive tree-search function (the one that visits all children, grandchildren) looks scary, but it visits each child once, so obviously takes just $O(n)$ time (where n is how many total children.) And we often use it on simple objects where we know there will be a fixed, small number of kids, so it's more like $O(1)$.

35.6 Other data structures

There are a few more list-like structures that are only useful because they have different big-O's for various things. This is just a very rough summary:

- B-tree. A clever way to have a root item with 2 children, which each have 2 children, and so on, making a big pyramid. Search, insert and delete are all $O(log(n))$. The drawback is you can't put things in a certain order – it's automatically sorted.
- K-tree, red/black tree: similar to B-trees. B-trees have 2 children, K-trees can have more. A red/black tree is a type of B-tree.
- Heap: a tree where finding the smallest item is $O(1)$ and insert/remove is $O(log(n))$. The items can't be in any order, and search takes $O(n)$. It's when for when you always want to handle the first thing in line. For example, storing Unity's waiting coroutines.

  A priority queue is often made using a heap (it's for taking items in order of highest priority, which may change.)

- Hash table. Search, insert and delete are $O(1)$! (on average.) Everything else is bad. There's no order, not even sorted. Finding the smallest is $O(n)$. Finding the next largest item in the list is $O(n)$. It takes up more space than a tree and requires extra set-up.

  C# calls theirs a Dictionary.

They're a little more complicated than that. Knowing the details and how and when to use them is why Com Sci grads get the big money.

35.7 Overview

Altogether, including ideas from the previous section on efficiency:

- In many places you don't care about how fast the code runs: things that rarely happen, that already have an on-purpose time-delay, or "how does this look?" code which you know will be either deleted or improved later.
- Many things run fast enough. A 2D tap-tap-tap puzzle game will run fast enough with easy-to-read code where you don't worry about speed and maybe do some "slow" things.

Assuming we're in the section where we think speed might matter:

- As you plan, think about arrays vs. linked lists, based on the big-O's of whatever you want to do. Generally, if you need to jump around a lot, you have to use an array. If you need to insert/delete lots from anywhere

except the end, you need a linked-list (you can fake changing the end of an array by using a bigger array than you need, with an extra `itemsUsed` variable.)

Maybe think about another fancy data structure (but not until you've tried them in a test project first, since nothing new ever works the way you thought it did.)

- As you write, think a little about the big-Os of things. If you have 50 monsters, each running an array loop which calls an $O(n)$ function, that's like $O(n^3)$. It might be worth trying to get that down.
- For arrays and lists, often sorting them, or keeping them sorted as you add new items is a big speed-up (instead of adding new items to the end, add them in the correct position. That takes longer, but you might search 1,000 times for every one time you add.)
- After you've gotten all the big-O speed-ups and want to look for small ones, use big-O estimates for where to start. For example, speed up the insides of nested loops before singles.

35.8 Numbers

This is the section with the numbers, which you can skip.

Measuring the time instructions take is fuzzy. Maybe adding takes 1 step, but adding floats takes longer than ints. But sometimes the computer does two things at once. And different CPUs can take different times.

`A[i]++;` is maybe 4 steps? Look up `i`, jump to that spot in `A`, add 1, then save it. But they might not all take the same time, so about 4 steps.

Things like square root and trig functions "take a long time," but not really. They might take 10 or 20 steps each. A big, fat trig-using math expression might take 80 steps. If you can get rid of some trig you don't need, you might get it down to 62.

This is what I mean by non-loops not mattering. 80 to 62 isn't a big speed-up compared to x100 faster, and 80 is a small number compared to a few thousand.

If we want to accurately measure the smallest array loop, it might take 7 steps. Just moving the loop is `i++` (2 steps?) and `i<A.Length` (3 steps?) and then `A[i]=0;` is 2 or 3 steps? The hypothetical simple 100 length loop is 700 steps total, making the 80-step math seem even smaller.

If we add a second line inside the loop, that doesn't really double the time. It's more like 7 steps increasing to 9. Put another way, cutting 2 lines inside a loop to 1 is really only a 20% speed-up.

If we put our 80-step math equation in a 1,000 step loop, of course it matters. We're taking 80,000 steps total. But reducing the middle to 62 steps is still less than a x2 speed-up. Removing the loop is still the best thing.

A worse big-O isn't always larger, but will always *eventually* be much larger. Suppose we have a single loop with 50 steps in it, and a sneaky nested loop with only 1 step in it. The times look like this:

```
size  single  nested
20  |   1,000       400
50  |   2,500     2,500
100 |   5,000    10,000
200 |  10,000    40,000
500 |  25,000   250,000
```

We might have a size 100 single loop and a size 20 nested which is currently much faster. But if they get larger, the nested loop quickly sucks up all the time.

And if you only have small arrays, you're probably not having speed issues anyway.

The other thing we have to worry about is working on the "fat" parts of our program, but big-O usually takes care of that:

Suppose our program has 10 equal-sized chunks. A 100x reduction of one means we still take 90.1% of the original time. But what really happens is the worse big-O stuff is also the longest, by a lot. For real, if one of those 10 parts has a nested loop, it's probably 90% of the time of the entire program.

In practice, the worst big-O functions are also the ones hogging up almost all of the time.