Chapter 43

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.

43.1 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++;
    }
```

Suppose every array is 1,000 long. Loop one runs 1,000 times, loop two runs a million times (`count1` is a million when this ends), and it runs `count2++` a billion times. Nested loops get very big, very quickly.

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'll never get a decent x100 speed up simplifying `if`'s and math, like we can simplifying loops.

- 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 make it a problem. A nested loop on 12 cats might be only 144 steps. But as the pet store grows we'll have 100 cats and the loops shoot up to 10,000 steps.
- It works well for functions that take lists as inputs. Say the function is ranked as a nested loop. It's safe to call with very small lists. If we need to call it with a big list, in another loop, at least we know this is a super-slow triple loop.
- 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 2D nested loop to set up a Tic-Tac-Tow board is always a mere 9 steps.

  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.

43.2 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;
```

Starting the loop at 1 is the logical thing to do, and also a tiny speed-up. But a loop that skips a few items still counts as a loop.

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. A "real" nested loop on 1,000 items runs a million steps. This runs only 1/2 a million if there are

no duplicates, and half that (on average) if there is one. But a quarter million is still only 4 times faster than a million, and way worse than 1,000 for a single loop. We count it as a standard nested loop.

Finding the index of the most common item is another nested loop:

```
int mostCommonItem(int[] A) {
  int mostNum=-999; // best count so far
  int mostIndex=-1;
  for(int i=0;i<A.Length;i++) {
    int cur=A[i];
    int count=howManyOf(A, cur);
    if(count>mostNum) { mostIndex=cur; mostNum=count; }
  }
  return mostIndex
}
```

This is a single loop around `howManyOf`, which is a loop. So it's a nested loop.

Inserting and removing items from arrays requires sliding everything after it (either sliding in to fill the hole, or sliding out to make room.) The `List` class has this built-in.

`L.Add(4)` takes 1 step. It adds to the end, so there's nothing to slide down. `L.Insert(0,4)` adds to the front, so is a full loop to make room. `L.Insert(i,4)`, where `i` is just some position is a half-loop on average, which counts as a loop.

All-in-all, adding or removing from the back is a non-loop, and anywhere else counts as a loop (in theory, only playing around with things at most 10 from the end would count as non-loops, but no one ever does that).

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 (not important, but about half if there are any, and the whole list when there aren't).

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's a binary search. There's 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 goes from a nested loop to a single loop (being sorted means identical items are together. We can count how many there are of an item in the same loop that looks at each different item. The code for this is later).

Sorting

Sometimes your 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.

43.2.1 Fun with bad loops

We know an easy way to make a reversed copy of a list is adding every item to the front of a new one, like this:

```
// copy L reversed into L2:
List<int> L2 = new List<int>();
for(int i=0; i<L.Count; i++)
  L2.Insert(0, L[i]);
```

But we know inserting to the front is a loop. This whole thing is a nested loop – a whooping million steps if `L` is length 1,000.

This version is a single loop:

```
List<int> L2 = new List<int>();
for(int i=L.Count-1; i>=0; i--)
  L2.Add(L[i]); // add to end is only one step
```

Removing all negative items looks like a simple loop. It goes back to front so that the remove-slide can't mess things up:

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

But `RemoveAt` anywhere except the end, is a loop. We can have negative items all over, so the whole thing is a nested loop (assuming a decent fraction of them are negative.)

It so happens that if you know you want multiple removes, you can redo it to use only a single loop. There's a built-in `RemoveAll` which is a single loop. It takes a function input for what to remove:

```
L.RemoveAll( (n)=>n<0 );
```

The loop runs through the array from start to back. When it finds something to remove, it starts sliding items 1 to the left, to fill the hole. But it also keeps checking. When it finds another item to remove, it increases the slide to 2:

```
int removedCount=0; // also how far to slide
// assume len is how much of the array we're currently using
for(int i=0; i<len; i++) {
  if(A[i]<0) { removedCount++; } // don't slide this down
  else A[i-removedCount]=A[i];
}
len-=removedCount;
```

In general, this is common: using a "do it once" method, over and over, can often be improved if we think about doing it to the whole list at once.

A standing-in-line type list is common – new items go in the back and are processed when they get to the front. It's often called a queue or first-in-first-out (FIFO).

Doing that with an array always has a loop. Depending on which way the line goes, we need to either add or remove from the front. With a Linked List we can use `L.AddLast(6);` and `n=L.First.Value` and `L.RemoveFirst()`, which all count as a single step.

43.3 Formal math

We really measure speed using simplified formulas, not loops. For example, a nested loop over a size n array takes $n * n$ steps. So we write it as $O(n^2)$. Once you get used to it, that's a shorter and more accurate way.

The rules are for writing Big-Oh notation:

- A capital O and parens goes around the whole thing. Besides standing for Big-Oh, it means "on the order of". $O(n^2)$ means: about n-squared steps.
- If there's no loop, write $O(1)$. It 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.

The terms for what we've been using, in increasing order of slowness:

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

$O(n^3)$ - triple nested loop

These are the terms you'll see listed in manuals. Inserting to the front of a `List` is $O(n)$, meaning it's a loop over all items. Inserting to the front of a Linked List is $O(1)$, meaning it's a few lines, no matter how long the list is.

If you're interested, Big-Oh started as a way to simplify the actual formula for how many steps. A function might do some work taking 50 steps, then run a loop over an array with 10 steps inside, then run a nested loop where the middle skips part. The actual formula is $50 + 10n + 0.5n^2$.

Big-Oh says we may as well reduce that to $O(n^2)$. That function basically takes nested loop time.

Using these terms: suppose we sped-up a loop from $6n$ to only $3n$. We know that's minor, since it's still a loop. The technical way is to say that is they're both $O(n)$; you didn't reduce the Big-Oh.

This is another detail if you're interested in the math: $O(n^{1.9})$ is also a category. There are a few horribly complicated algorithms with strange running times like that (with even stranger math proving it.)

If you graph $n^{1.9}$ and n^2, the second will pull further and further ahead. Eventually $n^{1.9}$ will be a thousand times faster. So that's great. But it might not happen until your arrays are a billion items or more.

The actual definition of Big-Oh is about being much faster *once the arrays get big enough*. How big is that? It depends. For most simple stuff, like going from a nested loop to a single loop, a few hundred items is enough to see a big speed-up.

Put another way, these rules are only useful with large lists, that you expect to grow. If method A takes 1,000 steps and method B takes 2,000, A is faster.

If A is a triple nested loop (and B isn't,) it's still faster. Big-Oh is only about asking yourself: "am I running a test array with 20 items? Will the real one have 200? If so, do I have nested loops that will balloon up?"

The last funny math note: sometimes you'll see two variables. Like a function taking 2 arrays might be $O(m * n)$. If you know one array will have 4 items at most, that's like $O(n)$. If both are about the same size, that's like $O(n^2)$.

43.4 More tricks and exceptions

Obviously, not all loops are problems, and not all problems are loops. We're really looking for things that take 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. It searches through all gameObjects. That's probably a big list, which will likely grow as you add a bigger map with more stuff in it. So it's a classic "it seems fast enough now, on this small list, but $O(n)$ is a warning it won't stay fast."

Some people like to use empty gameObjects as folders. Maybe all pick-ups go into an empty named `pickupHolder`. We can have lots of pick-ups, more and more as we grow the game. So now `transform.Find("healthPack5")` is like a real $O(n)$ function.

To see why that matters, suppose we do something with every health pickup, in a loop using Transform.Find:

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

We just accidentally wrote a slow nested loop. If we have 100 pick-upable items on the map, this takes about 5,000 steps. The purpose of Big-Oh is to help us spot that – an $O(n)$ function call in a loop. Yikes!

We could rewrite using a "go through children in order" loop:

```
foreach(Transform t in pickupHolder) // Unity's each child shortcut
  // grab name, reject non-healthpacks:
  string w=t.name;
  if(w.Substr(0,10)!="healthPack") continue;
  int packNum = ...
    ...
}
```

This is messier to write, but it's only a few hundred steps, compared to thousands. Made possible through big-Oh thinking.

Count the work, not the loops. Sometimes a nested loop just goes through the array once, so is really $O(n)$. This is the loop I promised to find the most common item in a sorted list:

```
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 fresh 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; }
}
```

The form is a nested loop. But if you trace it, both loops work together to push `i` one time through the array. It's really doing $O(n)$ work.

We've already seen the cut-in-half loop. On a size million array it runs 20 times, so we don't count that as a normal loop.

And it's possible to write a single loop that runs like two nested loops (at the end it would hand-move `i` and `j` the same way a nested loop would).

Funny loops like those are rare, and you can usually spot them easily enough by how they move the loop variable in odd ways.

Non-array loops count. Sometimes you loop over numbers. Suppose you check every angle from 0 to 180, going by 0.1. That's about 2,000 steps, more if you have to reduce the step for accuracy. Two of those could be nested, taking roughly 4 million steps. 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.)

For grids, it might make sense to have n be the length of a side (especially if it's mostly square,) or be the number of squares (which is better for long, narrow boards).

A function that touches every square once would be $O(n^2)$ when n is side-length, and $O(n)$ if n is how many actual squares.

If seems funny that we can just pick something, but we're only using it for comparison. Whatever we pick for n, we'll use that to measure every plan.

Recursion can be just about any Big-Oh. This function uses recursion to walk through an array, one step at a time, so it's $O(n)$:

```
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 from next item
}
```

Most recursive functions that call themselves only once are $O(n)$. But real recursive functions call themselves twice or more.

You can often logic those out: Flood fill hits every square once. A recursive tree search hits every child once; and sometimes you know the tree has at most 6 things, so the recursive function is essentially $O(1)$.

But recursion also can blast out of control with $O(2^n)$. That's exponential time (which is as bad as it sounds – worse than a mega-nested loop.)

The end result, which we already knew: recursion is a very tough topic.

43.5 Other data structures

Computer science has several ways to store data which are only useful because they have interesting Big-Oh's.

Many of these are built into things you already use, but it might be fun just to see the various things we use:

- B-trees. These are always sorted lists where search, insert and delete are all $O(log(n))$. The B stands for binary: they're really trees where everyone has 2 children, making a big pyramid. As a nice feature, they can also be stored in an array.

- K-trees, red/black trees. These are like B-trees. A red/black tree can have 1 or 2 children – it doesn't have to be a perfect pyramid, but has to be close. K-trees can have up to 3, 5 or 7 children (the K stands for how many they have.)

  Like other trees, most things are $O(log(n))$. Different trees are a little better or worse depending if you insert and remove more, or search more.

- Heaps. There are trees made so you can easily add items with an "importance value" and always pull out the most important. Most things are $O(log(n))$. That's all they're good for, but they're very good at it.
- Hash tables. Search, insert and delete are $O(1)$! (on average.) But everything else is terrible and it takes a little more space.

  These are how Maps and Dictionaries are made (ex: `A["cow"]=6;`).

None of these do anything an array can't do. They just do some things faster (and other things worse.) There's an entire course studying these things, and learning the Big-Oh's for everything so you know which one to use when.

43.6 Overview

Altogether, including ideas from the previous section on efficiency:

- In many places you don't care about how fast the code runs. For example, things that rarely happen, that purposely have a time-delay, or things we're just trying out.
- Many things run fast enough. A 2D tap-tap-tap puzzle game will run fine with triple-nested loops. If removing a loop would make it hard to read, it's not worth a X100 speed-up.

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.

  When you see a new language, look for the nice built-in array and linked-list types (in `C# List<int>` is really an array).
- As you write, think a little about the big-Oh's 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.
- Find the worst big-Oh. It probably sucks up 95% of the time. There's no point speeding up anything else.
- Think about sorting lists, then using the faster "only works if sorted" routines. If you have any nested loops, this is usually at least a x100 speed-up for that part.

  A little harder, you can add new items to the list in a way to keep it sorted. If you use the list a lot, and don't add or remove much, this can be fast.
- 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.

43.7 Numbers

This is the section with the numbers. Mostly explaining why measuring speeds is so fuzzy.

Adding two numbers might take 1 step, but adding floats takes longer than adding ints. Also, sometimes the computer can run two of your instructions at once. Different CPUs take different times for all of this.

`A[i]++;` might be 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 it's 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 inside. 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
```

Everything always charts out like this. More nested loops always eventually takes longer, then much longer, then much, much, much longer. The technical term for needing to get big enough is Asymptotic. As in "$O(n^2)$ is asymptotically much slower than $O(n)$".

If you only have small arrays, which will never get larger, you're probably not having speed issues anyway.