Binary search vs linear search revisited

If we take the square roots of all ints from 0 to \( n \), the UNIX utility `time` reports the following:

<table>
<thead>
<tr>
<th>( n )</th>
<th>Binary search time (s)</th>
<th>Linear search time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>0.139</td>
<td>1.839</td>
</tr>
<tr>
<td>2,000,000</td>
<td>0.282</td>
<td>5.193</td>
</tr>
<tr>
<td>3,000,000</td>
<td>0.423</td>
<td>9.544</td>
</tr>
<tr>
<td>4,000,000</td>
<td>0.560</td>
<td>14.691</td>
</tr>
<tr>
<td>5,000,000</td>
<td>0.698</td>
<td>20.529</td>
</tr>
</tbody>
</table>

You’ll note that these growth rates don’t seem quite in line with the big-\( O \) that binary and linear search should have. Let’s briefly examine why.

We’re looping from \( i = 0 \) to \( n \) in both cases, running an operation that takes time proportional to \( i \).

So, binary search should take time roughly:

\[
\sum_{i=0}^{n} \log(i) = \frac{\log^2(n) + \log(n)}{2} \in O(\log^2(n))
\]

And linear search takes roughly:

\[
\sum_{i=0}^{n} i = \frac{n^2 + n}{2} \in O(n^2)
\]

These times are much more in line with what we’re seeing. Note that the times for binary search are also off due to a bug in my implementation, where I always started the upper bound at a constant number (so we were searching a fixed range of values always).

Here’s a slightly easier example, when we take the square root of \( n \) 5,000,000 times for the linear search, and 100,000,000 times for binary search:

<table>
<thead>
<tr>
<th>( n )</th>
<th>Binary search time (100,000,000 times)</th>
<th>Linear search time (5,000,000 times)</th>
<th>isqrt(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>8.658 (6 iterations)</td>
<td>14.288 (1001 iterations)</td>
<td>1000</td>
</tr>
<tr>
<td>2,000,000</td>
<td>10.886 (10 iterations)</td>
<td>20.205 (1415 iterations)</td>
<td>1414</td>
</tr>
<tr>
<td>3,000,000</td>
<td>10.171 (9 iterations)</td>
<td>24.735 (1733 iterations)</td>
<td>1732</td>
</tr>
<tr>
<td>4,000,000</td>
<td>8.746 (6 iterations)</td>
<td>28.555 (2001 iterations)</td>
<td>2000</td>
</tr>
<tr>
<td>5,000,000</td>
<td>11.308 (10 iterations)</td>
<td>31.917 (2237 iterations)</td>
<td>2236</td>
</tr>
<tr>
<td>6,000,000</td>
<td>14.183 (12 iterations)</td>
<td>34.939 (2450 iterations)</td>
<td>2449</td>
</tr>
<tr>
<td>7,000,000</td>
<td>12.748 (12 iterations)</td>
<td>37.744 (2646 iterations)</td>
<td>2645</td>
</tr>
<tr>
<td>8,000,000</td>
<td>11.187 (10 iterations)</td>
<td>40.355 (2829 iterations)</td>
<td>2828</td>
</tr>
<tr>
<td>9,000,000</td>
<td>10.567 (8 iterations)</td>
<td>42.795 (3001 iterations)</td>
<td>3000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>11.812 (11 iterations)</td>
<td>45.106 (3163 iterations)</td>
<td>3162</td>
</tr>
</tbody>
</table>
Big-O

I used the $\in$ symbol above for big-O, which might be confusing if you’ve never dealt with big-O formally before. So, let’s familiarize ourselves with the formal definitions.

First of all, $O(g(n))$ is a set of functions. The formal definition is:

$$f(n) \in O(g(n)) \text{ if and only if there is some } c \in \mathbb{R}^+ \text{ and some } n_0 \in \mathbb{R} \text{ such that for all } n > n_0, f(n) \leq c \cdot g(n).$$

(Note that there should be some absolute values in there, but as we don’t generally deal with negative runtime complexities it’s fine to neglect them for this class.)

That’s a bit formal and possibly confusing, so let’s look at the intuition. Here’s a diagram, courtesy of Wikipedia. (Note that it uses $x$ and $x_0$ instead of $n$ and $n_0$.)

Below $n_0$, the functions can do anything. Above it, we know that $c \cdot g(n)$ is always greater than $f(n)$.

In problems 1 and 2, we play with the definition of big-O.

Something that’s very important to note about this diagram is that there are infinitely many functions that are in $O(g(n))$: If $f(n) \in O(g(n))$, then $\frac{1}{2} f(n) \in O(g(n))$ and $\frac{1}{4} f(n) \in O(g(n))$ and $2 f(n) \in O(g(n))$. In general, for any constant $k$, $k \cdot f(n) \in O(g(n))$. In problem 3, we prove this.

Something that will come up often with big-O is the idea of a tight lower bound.

It’s technically correct to say that binary search, which takes around $\log(n)$ steps on an array of length $n$, is $O(n!)$, since $n! > \log(n)$ for most $n$, but it’s not very useful. If we ask for a tight bound, we want the closest bound you can give. For binary search, $O(\log(n))$ is a tight bound because no function that grows more slowly than $\log(n)$ provides a correct upper bound for binary search.

Unit testing

Unit testing is key to being able be confident about code correctness and to helping you debug code when things go wrong.

Jonathan Clark, another 122 TA, prepared an excellent handout that goes into depth on unit testing. It’s available at [http://www.cs.cmu.edu/~jhclark/courses/15122-f12/testing_recitation/testing.pdf](http://www.cs.cmu.edu/~jhclark/courses/15122-f12/testing_recitation/testing.pdf) I won’t repeat it in full here, but there are a few crucial takeaways.

When testing your code, you should start with the assumption that it’s incorrect for some input, and that you want to find that input. It’s critical to think about corner cases and edge cases — those are things that are out of the ordinary, but are allowed by the preconditions of your function. For instance, if your function operates on ints, some edge cases include INT_MIN, INT_MAX, 0, -1, and 1.
If you’re trying to test code, it’s critical to come at it with the attitude that you want to break it. Pretend your worst enemy wrote the code, and that you want to show them all of the problems with it to get them back for that thing they did. (This is necessary because in the real world you will have to deal with edge cases: if you don’t, your code will break for at least some people, possibly making whatever you wrote unusable for them.)

Now, let’s do problem 4. I’m your worst enemy, I have some code, and you want to break it so we can fix it.

**Sorting**

Sorting is a really useful task that’s central to many programs (for instance, spreadsheet programs) as well as many algorithms (for instance, Google PageRank needs to sort pages before presenting them).

One such algorithm is selection sort, which we discussed in lecture. The basic idea behind selection sort is that we find the smallest number in the region of the array that we’re considering, switch that with the first element of the region of the array that we’re switching, shrink the size of the array we’re considering by one, and repeat until we’re considering no elements of our array.

Here’s a good visualization of selection sort on a series of poles: [http://www.youtube.com/watch?v=LuANFAXgQEW](http://www.youtube.com/watch?v=LuANFAXgQEW).

In that video, the light blue square points to the smallest element we’ve found so far, the purple square points to the element we’re comparing with the minimum, and the pink rectangle is the region of the array we still need to consider.

This is not a particularly efficient algorithm: if we have $i$ elements in the region of the array we’re considering, we need to look at all $i$ of them, every time.

So, the overall amount of work we’ll have to do is $n + (n - 1) + (n - 2) + \cdots + 2 + 1$, or:

$$\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}$$

Now that you know big-O notation and have started to learn about time complexity, it should be possible for you to figure out what the time complexity of this algorithm is in big-O notation.

```java
int min_index(int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures le_seg(A[\result], A, lower, upper);
{
    int m = lower;
    int min = A[lower];
    for (int i = lower+1; i < upper; i++)
    //@loop_invariant lower < i && i <= upper;
    //@loop_invariant le_seg(min, A, lower, i);
    //@loop_invariant lower <= m && m < upper;
    //@loop_invariant A[m] == min;
    {
        if (A[i] < min) {
            m = i;
            min = A[i];
        }
    }
    return m;
```
20 }
21 void sort(int[] A, int n)
22 //@requires 0 <= n && n <= length(A);
23 //@ensures is_sorted(A, 0, n);
24 {
25     for (int i = 0; i < n; i++)
26         //@loop_invariant 0 <= i && i <= n;
27         //@loop_invariant is_sorted(A, 0, i);
28         //@loop_invariant le_segs(A, 0, i, i, n);
29         {
30             int m = min_index(A, i, n);
31             //@assert le_seg(A[m], A, i, n);
32             swap(A, i, m);
33         }
34     return;
35 }

Practice!

1. Rank these big-O sets from left to right such that every big-O is a subset of everything to the right of it. (For instance, \(O(n)\) goes farther to the left than \(O(n!)\) because \(O(n) \subset O(n!)\).) If two sets are the same, put them on top of each other.

\[
\begin{align*}
O(n!) & \quad O(n) \quad O(4) \quad O(n \log(n)) \quad O(4n+3) \quad O(n^2 + 2n + 3) \quad O(1) \quad O(n^2) \quad O(2^n) \\
O(n^3 + 300n^2) & \quad O(\log(n)) \quad O(\log^2(n)) \quad O(\log(\log(n))) \quad O(n^3)
\end{align*}
\]

2. Using the formal definition of big-O, prove that \(n^3 + 300n^2 \in O(n^3)\).

3. Using the formal definition of big-O, prove that if \(f(n) \in O(g(n))\), then \(k \cdot f(n) \in O(g(n))\) for \(k \geq 0\).

One interesting consequence of this is that \(O(\log_i(n)) = O(\log_j(n))\) for all \(i\) and \(j\) (as long as they're both greater than 0), because of the change of base formula:

\[
\log_i(n) = \frac{\log(n)}{\log(i)}
\]

But \(\frac{1}{\log(i)}\) is just a constant! So, it doesn't matter what base we use for logarithms in big-O notation.

4. Let's work together! I have a few functions, with known specs. Let's try to break them without looking at the source code, then use that knowledge we gain from breaking them to fix them. First we'll do safe_add, and then we'll do evil_sort if there's time.

5. Prove that min_index is correct.

6. Prove that sort is correct.