Algorithm Efficiency and Correctness

- At this point, we’ve seen a number of examples on how we can successfully develop algorithms to get certain tasks done

- We’ve also seen that there may be more than one algorithm to accomplish the same task, such as searching and sorting

- So what’s the difference other than the algorithm itself? How would we choose one over the other? Can we make this choice objective or quantitative?

A Little Exploration

Consider a comparison between sequential and binary search — suppose we have a million records to search

- With sequential search, on average we would search half of the list — 500,000 records — before we find what we’re looking for

- If the item we’re looking for is the first one in the list, then we pull a single record; if it’s the last one, then we end up pulling all one million records

- With binary search, we look at one record — the middle one — per “turn,” then split the list in half; so, we go from a million to 500,000...250,000...125,000...62,500...31,250...15,625...7,812...3,906...1,953...976...488...244...122...61...30...15...7...3...1...that’s a maximum of 20 “turns” to go through all million records

- For binary search, we pull a single record if the one we’re looking for is in the middle of the list (instead of the first), and we pull the maximum of 20 if the item we’re looking for requires a reduction all the way to a single-item list — note that we may find it sooner
How “Big” is an Algorithm?

• The exercise that we just went through — thinking about how many steps we would have to take in order to accomplish sequential vs. binary search — falls under the area of algorithm analysis

• Algorithm analysis tries to “measure” an algorithm; when it comes to the “turns” that we looked at, the number of “turns” translates into time — one of the major measurements for an algorithm

• Another major measure is space — how much memory does an algorithm need?

• An important element of algorithm analysis is that we must look at our algorithm in general — that is, over a wide range of instances
  ◇ For searching, this translates to thinking about all possible combinations of lists and search targets that we may encounter when using the algorithm

• As a first cut, we note that there are best, worst, and average cases for an algorithm — and that these situations may vary widely

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Search</td>
<td>First item in the list: one “turn”</td>
<td>Last item in the list: one million “turns”</td>
<td>Middle of the list: 500,000 “turns”</td>
</tr>
<tr>
<td>Binary Search</td>
<td>Middle of the list: one “turn”</td>
<td>Reduce list to one or less: 20 “turns”</td>
<td>Halfway through the reduction: 10 “turns”</td>
</tr>
</tbody>
</table>

◇ Note that the table is specific to a list of one million items — so still not quite general
◇ Still, the table strongly suggests that binary search is the faster algorithm overall, in general
How Does an Algorithm “Grow?”

- What if, instead of supposing that we have a list of one million items, we have a list of \( n \) items? This truly generalizes our analysis of the algorithm...

- Note that it isn’t too hard to replace “one million” with \( n \) in the previous table

<table>
<thead>
<tr>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>First item in the list: one “turn”</td>
<td>Last item in the list: ( n ) “turns”</td>
<td>Middle of the list: ( n / 2 ) “turns”</td>
</tr>
<tr>
<td>Binary Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Middle of the list: one “turn”</td>
<td>Reduce list to one or less: ( \log_2 n ) “turns”</td>
<td>Halfway through: ( \log_2 (n / 2) ) “turns”</td>
</tr>
</tbody>
</table>

- Let’s try this exercise with insertion sort:
  - Insertion sort’s best case is a list that is already sorted, since we just perform a single comparison (to the item above the “hole”) for each pivot position — thus, for a list of \( n \) items, we would make \( n - 1 \) comparisons (since we start with the second item in the list)
  - At worst, the pivot always has to move to the top of the list — this would happen if the list is in reverse order — and so pivot position \( i \) would have to make \( i \) comparisons (since it is \( i \) slots away from the top)
  - Since we still go through each pivot position, we would make \( 1 \) comparison for pivot 1, 2 comparisons for pivot 2, etc. — the net result is \( 1 + 2 + 3 + \ldots + n - 1 \), which is the summation from 1 to \( n - 1 \), or (following the well-known formula) \( (n - 1)n / 2 \)
  - For the average case, we just figure that we move halfway to the top for each pivot position; this is just half of the worst-case summation, so we get \( (n - 1)n / 4 \)

- The analysis for quicksort is much more involved, but it has been done and these are the results:
  - In the best case, we get \( n \log_2 n \), with the average case coming in at approximately \( 2n \log_2 n \) — as with binary search, the \( \log_2 n \) comes from repeatedly splitting the list in half
  - At worst, quicksort performs at around \( n^2 \) — this is when, instead of splitting the list in half, we always have the pivot at the first element (resulting in no splits at all!)
If we take a pessimistic view and focus on the worst-case scenario, then we can have some kind of guarantee that, no matter what, an insertion sort will never take longer than \((n - 1)n / 2\) comparisons.

Thus, we can say that, at worst, it will take 45 comparisons to sort a list of ten items, 4,950 comparisons to sort 100 items, and 499,999,500,000 comparisons to sort one million items!

The numbers are interesting, but there’s a better way to see how the algorithm grows:

```
<table>
<thead>
<tr>
<th>Number of items in list</th>
<th>Time to perform algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insertion sort</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
</tr>
</tbody>
</table>
```

Graph showing the comparison between Insertion sort and Quicksort.
Big Theta (Θ): Efficiency by Shape

- Note how the “shape” of the graphs gives us an idea of how an algorithm fares depending on the “size” of the problem that it is solving — in the case of sorting and searching, this involves the number of items in the list.

- Other algorithms may have different measures for “problem size;” for example, make change may vary by the number and value of coin denominations available.

- In any case, computer science has a standard notation for characterizing the efficiency of an algorithm for a problem of size n — this is “big theta” or simply Θ.

- Big theta notation is written as $\Theta(expr)$, where expr is a mathematical expression written in terms of $n$.

- Further, we usually eliminate constant coefficients and addends, since these don’t typically contribute much in how an algorithm grows (or deteriorates):
  - We can therefore have insertion sort being $\Theta(n^2)$.
  - Quicksort is $\Theta(n \log_2 n)$ on average and $\Theta(n^2)$ at worst.
  - Looking back on our search algorithms, we can then say that sequential search is $\Theta(n)$ and binary search is $\Theta(\log_2 n)$.

- Comparing how $expr$ changes with respect to $n$ thus allows us to compare algorithms in a quantitative way.

- $\Theta(expr)$ is typically read as “big theta of $expr$” or “order of $expr$” — leading to an alternative (less formal) name for the notation: “big O”.
Correctness and Verification

• It’s one thing to design an algorithm and believe that it works correctly…but is there a way to know that it does work correctly?

• This is another wide open area within computer science: formal verification of the correctness of an algorithm or of a computer program that claims to execute an algorithm

• The task resembles mathematical proof in many ways; in fact, there are those who might argue that there is no difference at all

• One approach to verification flows in this way:
  ◇ Specify the initial assumptions (or preconditions) that are taken to be true at the start of a program (note the analogy to mathematical axioms or definitions)
  ◇ Note that actions in an algorithm or program (assignments, conditions, loops, recursion) may gradually transform these preconditions into other states or effects
  ◇ At a given point in an algorithm, one then makes assertions of what should be true at that moment; the algorithm is correct if each assertion can be shown to always hold true

• Formal verification methods have not yet seen much adoption or use; thus, bugs continue to abound in software, requiring extensive [manual] testing

• For the moment, many software projects use automated unit tests to (partially) verify correctness
  ◇ A unit test is a program that tests another program; the program being tested performs certain functions, and the unit test makes assertions on what must be true after each task
  ◇ Unit tests are only as thorough as the programmer who wrote them — they help validate certain situations but do not constitute absolute proof…still they’re better than nothing