Introduction to Computational Complexity

Simply put, *computational complexity* is a measure of the work (i.e. time – more specifically, computational operations) or space required for a given solution method for a stated problem. *Computational complexity theory* is the branch of computer science that focuses on classifying problems according to the computational complexity of their solutions. We thus use the term when discussing a specific algorithm, a class of algorithms, or even *all* algorithms for solving a given problem.

Example: Searching an unsorted array

Imagine we have an unsorted sequence (e.g. array or list) of n elements, $e_1, e_2, \ldots, e_n$. To determine whether a specific element or value e_s is contained in the sequence, we generally have no choice but to iterate over it until we find e_s; if we reach the end of the array without finding it, then it isn't present. This is called a *linear search*.

- Given an input value e_s, and an unordered sequence S with the elements $e_1, e_2, \ldots, e_n$.
- For each e_i in S:
    - If e_i = e_s, the item is found, and there is no need to continue. Return i, the position in the sequence where e_s was found.
- Return 0, indicating that e_s was not found in S.

Questions

1. If $e_s \in S$:
    - What is the smallest number of comparisons the algorithm above will make, before returning?
    - What is the largest number of comparisons that can be made?
    - What is the expected number of comparisons?
2. If $e_s \notin S$:
    - What is the smallest number of comparisons the algorithm above will make, before returning?
    - What is the largest number of comparisons?
    - What is the expected number of comparisons?

3. What is the time complexity of a linear search?

4. What is the space complexity (i.e. additional memory used) of a linear search?

In both of the above cases, we end up with an expected value for the number of comparisons (i.e. the time complexity) that's a linear function of n: for $e_s \in S$, we have $\frac{n+1}{2}$; for $e_s \notin S$, we have n.

In many cases, we'd like to be able to discuss the computational complexity in even more general terms, ignoring coefficients and constant additive terms. Thus, we might say that whether or not $e_s \in S$, the time complexity of the linear search is "on the order of n". (If, instead of a linear function of n, we had a second-order polynomial, we would say that the time complexity is on the order of n^2, and so on.)

Prior to the advent of computer science, this concept was used in analytic number theory, where we might wish to characterize the behavior of some function in terms of bounds expressed as simpler functions. In the late 19th century, number theorist Paul Bachmann introduced the shorthand $O(\ldots)$ for "order of". Thus, in our example, we would say that the time complexity of a linear search is $O(n)$.

In the 1970s. Donald Knuth introduced additional computer-science-specific definitions and distinctions for these and other computational complexity concepts. As part of that, he referred to the O symbol as the upper-case Greek letter omicron, rather than the Latin "O". He also provided this formal definition:

$$f(n) = O(g(n)) \iff \exists k > 0, \exists n_0 : \forall n > n_0, |f(n)| < k \cdot g(n)$$

In English, we can state this as

> The function $f(x)$ is of the order $g(x)$, if and only there exists some $k > 0$, and some n_0, such that for all $n > n_0$, $|f(n)| < k \cdot g(n)$.

Oversimplifying a bit, we can think of this definition as stating that if the number of operations (or the space required) for an algorithm increases as the size of the problem n increases, in such a way that over the long run, it is less than some function $g(n)$, multiplied by a positive constant, then the computational complexity of the algorithm is $O(g(n))$.

By itself, this may or may not be very informative: $k \cdot g(n)$ might in fact be far above $f(n)$, and the definition still holds, but that doesn't tell us much about $f(n)$. When we combine the $O(g(n))$ upper bound with the $\Omega(g(n))$ lower bound – which we might express together as a $\Theta(g(n))$ – we start to have something very powerful. Nonetheless, $O(\ldots)$, or "big O" notation, is a widely used shorthand for talking about computational complexity of algorithms.

Example: Searching a sorted array via binary search

Imagine we now have a *sorted* sequence of n elements, $e_1, e_2, \ldots, e_n$, where there is some ordering function d, such that $d(e_i) \leq d(e_{i+1}), \forall i = 1 \ldots n-1$ (i.e. the list is in ascending order, based on the function d). Further, assume that the for any e_i and e_j, $d(e_i) = d(e_j) \iff e_i = e_j$; that is, two elements of the sequence are considered equal if and only if the function d gives the same value for both. Finally, assume that we don't necessarily have to access the elements in order, but can access any item in essentially constant time; that is, it does not take significantly more time to access e_i than to access e_1, for any i in $2 \ldots n$.

To determine whether a specific element or value e_s is contained in the sequence, we now have a powerful alternative to linear search: Given any element e_j, if $d(e_j) > d(e_s)$, then we know that if $e_s \in S$, it must be located before e_j in the sequence, if it's actually in the sequence at all; on the other hand, if $d(e_j) < d(e_s)$, then e_s must be located after e_j in the sequence (if at all). If we pick e_j carefully – in this case, we pick it right in the middle of the portion of the sequence we're searching – then we divide the sequence in half each time we select and compare a new e_j to e_s. This is called a *binary search*.

- Given
    - The input value e_s, to search for.
    - The ordering function d.
    - The sequence S with the elements $e_1, e_2, \ldots, e_n$, ordered such that $d(e_i) \leq d(e_{i+1}), \forall i = 1 \ldots n-1$.
- Let $a = 1$.
- Let $b = n$.
- While $a \leq b$:
    - Let $j = \lfloor \frac{b-a}{2} \rfloor$.
    - If $d(e_j) = d(e_s)$:
        - Found! Return j, the position in the sequence where e_s is located.
    - If $d(e_j) > d(e_s)$:
        - Let $b = j - 1$.

    Otherwise, if $d(e_j) < d(e_s)$:
        - Let $a = j + 1$.
- Return $-a$, which is the negative of the position where e_s could be inserted into S to preserve

its ordering.

Note that we could state the above algorithm recursively, as follows:

To search a range of a sequence for an item,

- If the range to search is empty, then it doesn't contain the searched-for item; otherwise, compare the searched-for item to the item at the midpoint of the range.
- If the searched-for item isn't found at the midpoint, then use the comparison to determine which half-range (above or below the midpoint) should become the new search range.
- Search the new range for the item.

Questions

1. If $e_s \in S$:
    - What is the smallest number of comparisons the algorithm above will make, before returning?
    - What is the largest number of comparisons that can be made?
    - What is the expected number of comparisons?
2. If $e_s \notin S$:
    - What is the smallest number of comparisons the algorithm above will make, before returning?
    - What is the largest number of comparisons?
    - What is the expected number of comparisons?
3. What is the time complexity of a binary search?
4. What is the space complexity of a binary search?

Growth in search space vs. solution algorithm

Note that the size of the search space for a solution to a problem may grow much more quickly than the size of the problem inputs – but that doesn't mean that the work required for the solution algorithm grows at the same rate as the solution search space.

For example, consider sorting a sequence of distinctly valued items. If there are n items, then there are $n!$ distinct orderings of those items – only one of which would be considered sorted (assuming we are restricting ourselves to ordering by ascending value). However, we would be very foolish to search for

the sorted order by brute force, iterating over the $n!$ possible orderings until one is found that satisfies the order relationship across all of its items. Instead we will probably employ one of the widely used sorting algorithms (merge sort, quick sort, heap sort, etc.), which typically have time complexity of $O(n \log n)$ – which is *much* smaller than $O(n!)$.

On the other hand, we can see that *verifying* whether a given sequence is in order is quite simple: compare each item in turn to the item following it; if $d(e_i) \leq d(e_{i+1})$, for $i = 1 \ldots n-1$, then the sequence is in ascending order. This verification uses $(n-1)$ comparisons and thus has time complexity $O(n)$, which is less than the work required to perform the sort itself. In fact, there are many cases where verification of a solution is less difficult than finding the solution.

Example: Merge sort

The recursive algorithm for a merge sort is as follows:

To sort a sequence S containing n items, using the ordering function d:

- If $n = 1$, the sequence is already sorted; return.
- Divide the sequence into 2 subsequences: S_1, containing the first $\lfloor \frac{n}{2} \rfloor$ elements of S; and S_2, containing the remaining elements of S.
- Sort S_1 and S_2 separately (recursive invocation).

(The remaining steps constitute the *merge phase* of the algorithm.)

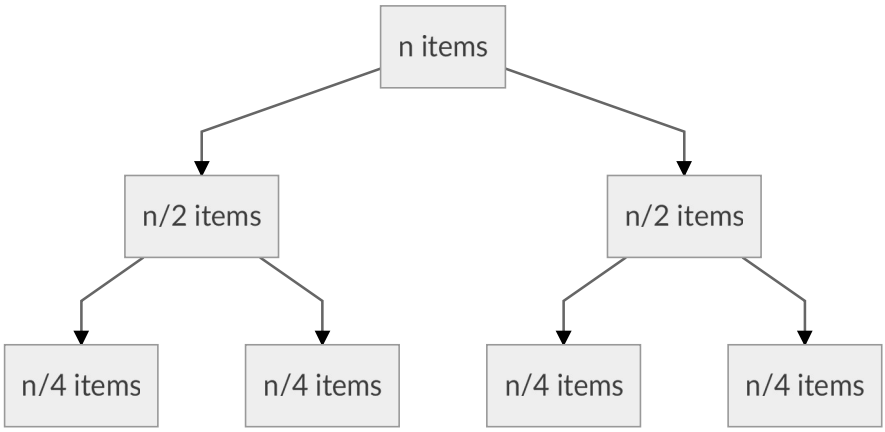
- Create an empty sequence S_c.
- While S_1 and S_2 contain elements:
    - Compare the first elements of S_1 and S_2 using d, to find the lower-valued of the two.
    - Remove the lower-valued element from its respective sequence, and append it to S_c.
- Append all remaining elements of S_1 or S_2 (whichever is not empty) to S_c, preserving their order.
- Replace the contents of S with the contents of S_c.

Questions

1. What is the time complexity of the merge phase of the algorithm? (It may help to think of it in terms of the minimum and maximum number of comparisons we might need to make between

two sequences, each originally of size $\frac{n}{2}$, in order to produce the merged sequence of size n.)

2. Each time we split a sequence into 2 smaller sub-sequences of equal (or nearly equal) size, we can think of this as a branch in a tree:



   In the example above, the tree is 3 levels deep. If we start with a sequence of n items, how many levels deep will the tree get, until all of its leaves (terminal node, from which no branches emanate) have a single item?

3. Can we use the answers to 1 & 2 to estimate the time complexity of merge sort?

4. What is the space complexity of merge sort? That is, how much additional memory (in "big O" terms) is required, beyond the original sequence itself? (For a recursive algorithm, don't forget that each recursive invocation will generally require stack space.)

Analyzing time complexity of recursive algorithms

Some algorithms are most easily expressed in a recursive form (though we may not necessarily implement them that way; remember that recursion can *always* be translated into iteration). In addition to (sometimes) making it easier to verify the correctness of an implementation, a recursive algorithm gives us some powerful mathematical tools for examining the computational complexity of the algorithm.

One of these tools is the *Master Theorem* (also known as the *Master Method*). To use this, we express the time complexity for executing an algorithm on n items in terms of the time complexity for executing recursively on the smaller subproblems, along with the time complexity for creating the smaller subproblems and then combining the solutions from the subproblems into a solution to the overall problem:

$$T(n) = a \cdot T(\frac{n}{b}) + O(n^d)$$

where

$$\begin{aligned} T(n) &= \text{time complexity of the algorithm for a problem of size } n. \\ a &= \text{number of recursive calls.} \\ b &= \text{input shrinkage factor.} \\ d &= \text{exponent in time complexity of split/combine step.} \end{aligned}$$

If we can express $T(n)$ as above, then the Master Theorem instructs us to compute $(b^d - a)$, and use the result as a discriminant to find $T(n)$:

$$T(n) = \begin{cases} O(n^d \log n), & \text{if } b^d - a = 0. \\ O(n^d), & \text{if } b^d - a > 0. \\ O(n^{\log_b a}), & \text{if } b^d - a < 0. \end{cases}$$

Example: Using the Master Theorem to estimate complexity of binary search

Each time we split the sequence S (with n elements) into 2 sub-sequences, we make 1 recursive call, to search the relevant sub-sequence with $\frac{n}{2}$ elements. The time complexity for splitting the sequence is constant (it doesn't depend on the size of the sequence); similarly, when the recursive call completes, it returns the final search result, so there is essentially no work expended after that point. So, for binary search, we have,

$$\begin{aligned} a &= 1 \\ b &= 2 \\ d &= 0 \end{aligned}$$

Here, $(b^d - a) = (2^0 - 1) = 0$, so from the above,
$T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$. Thus, the time complexity of binary search is $O(\log n)$.

Example: Using the Master Theorem to estimate complexity of merge sort

Each time we split the sequence S (with n elements) into 2 sub-sequences, we make 2 recursive calls,

to sort sequences with $\frac{n}{2}$ elements. The time complexity for splitting the sequence is constant (it doesn't depend on the size of the sequence), but re-combining the sequences after both are sorted is $O(n)$. Thus, for merge sort,

$$a = 2$$
$$b = 2$$
$$d = 1$$

Here, $(b^d - a) = (2^1 - 2) = 0$, so from the above,
$T(n) = O(n^d \log n) = O(n^1 \log n) = O(n \log n)$. Thus, the time complexity of merge sort is $O(n \log n)$.