Divide and Conquer Algorithms : Steps for Design and Analysis

Suitable Problems Divide-and-conquer usually improves the polynomial in the runtime of the obvious algorithm, rather than making an exponential-time algorithm polynomial. However, that is not universal, and occasionally a divide-and-conquer algorithm is poly-time even when the obvious algorithm is exponential.

Even if the divide-and-conquer algorithm has the same asymptotic time complexity as the obvious algorithm, it may still be a win in certain circumstances. First, divide-and-conquer algorithms typically have better locality of reference than non-divide-and-conquer versions, so there may be a win in memory access times. Secondly, divide-and-conquer algorithms are usually highly parallelizable, so if you have multiple processors available, it may be a good idea to use a dnc algorithm.

Here's a partial list of common situations in which a divide-and-conquer approach helps:

A. Divide-and-conquer can help when a problem involves a group of operations that are correlated or repeated. Using divide-and-conquer can speed these algorithms by doing computations once, rather than re-doing them for several different parts. For example, finding all consecutive sums in an array the naive way involves $\Theta(n^3)$ additions, but many of the sums involve adding many of the same array elements. We saw a divide-and-conquer algorithm that solved the same problem in $\Theta(n^2)$ time, because it reused the work done in the sub-problems in the larger problems. Strassen matrix multiplication is a more subtle example, where the correlation is only apparant in retrospect (if that).

B. Divide-and-conquer can help when a problem involves a decision that, once made, fractures the problem into disjoint pieces. This is especially true if some of the pieces will become irrelevant after the decision. For example, in binary search, the decision is "Does the given element occur before or after the median element?" If the answer is before, the list after the median becomes irrelevant, and vice versa. In isomorphism of binary trees, the decision is "does the left child of tree 1's root map to the left or right child of tree 2?". If it is mapped to the left, the right sub-tree of tree 2 becomes irrelevant to the mapping for the left sub-tree of tree 1; and symmetrically for the right sub-tree.

These are two common cases when DnC is useful, but not an exhaustive list of all possibilites.

Basic Design Steps

1. Figure out what a "sub-problem" is. It might be a part of the input, but it might also be how the problem would simplify if a certain fact about the solution were known, as in the second paradigm above. Usually, once you know what the sub-problems are, it is straight-forward to show how to compute them from the main problem. This is the "divide" part. Note that, in order to guarantee polynomial-time, each sub-problem must be at most size n/b, where b is a constant greater than 1. Furtermore, there should be at most a constant number of sub-problems.
2. Make a recursive call for each sub-problem. Alternatively, do what calculations you need to decide WHICH candidate sub-problems are needed, and only do the recursive calls for necessary sub-problems. Try to make the total number of recursive calls, and sizes of sub-problems, as small as possible.
3. How do the solutions for the sub-problem help find the actual solution? The "conquer" stage combines the sub-problems, uses them to fill in missing parts, or otherwise calculates the solution to the main problem from the recursive calls.

Time analysis: regular case The simplest case of a divide-and-conquer to analyze is the regular case, where each of a sub-problems has size at most n/b. Then the time satisfies the recursion: $T(n) = aT(n/b) + f(n)$, where $f(n)$ is the total time in the non-recursive parts, the "divide" part and the "conquer" part. Then we have three possibilities:

1. Top-heavy: Most of the work is being done in the top few levels of the recursion. Thus, the total time is $T(n) \in O(f(n))$.
2. Steady-state: About equal amounts of work are being done at each of the $\log_b n$ levels of the recursion. Thus, the total time is $T(n) \in O(f(n) \log n)$, since $b > 1$ is a constant.
3. Bottom-heavy: Most of the work is being done in the $a^{\log_b n} = n^{\log_b a}$ base cases of the recursion. Thus, $T(n) \in O(n^{\log_b a})$.

If $f(n) \in \Theta(n^k)$ for some constant k, there is a simple calculation to decide which case we are in. The i'th level of the recursion will have

a^i sub-problems, each of size n/b^i. So the total time for this level is roughly $a^i(n/b^i)^k = n^k(a/b^k)^i$. Thus,

1. If $a < b^k$ then the work is decreasing exponentially, so we are in the top-heavy case, $T(n) \in \Theta(n^k)$.
2. If $a = b^k$ then the work is staying the same, so we are in the steady-state case $T(n) \in \Theta(n^k \log n)$.
3. If $a > b^k$ then the work is increasing exponentially, so we are in the bottom-heavy case $T(n) \in \Theta(n^{\log_b a})$.

If $f(n) \in \Theta(n^k g(n))$, where $g(n) \in n^{o(1)}$ is a monotone and convex function, then typically $g(n/b) \in (1 + o(1))g(n)$. If this is true, then the first and last cases above remain bottom-heavy and top-heavy, respectively. If however, $a = b^k$, any of the three cases could pertain, or we could even have something between the cases, where work is growing or shrinking, but not at the rate so that the middle is negligible compared to the top or bottom. We then need to revisit our reasoning above to sum up all the different levels of the recursion, or at least get a bound.

For example, let $T(n) = 2T(n/2) + f(n)$ where $f(n) = n2^{\sqrt{\log n}}$. In the i'th level of the recursion, we have 2^i sub-problems of size $n/2^i$. So the work at that level is $Work_i = 2^i(n/2^i)2^{\sqrt{\log(n/2^i)}} = n2^{\sqrt{\log n - i}} = f(n)2^{\sqrt{\log n - i} - \sqrt{\log n}} = f(n)2^{-i/(\sqrt{\log n - i} + \sqrt{\log n})}. = n^{1+1/(\log(\log n - i))}2^{-i/\log(\log n - i)}$. Thus, we can bound $Work_i$ by $f(n)2^{-i/\sqrt{\log n}} \leq Work_i \leq f(n)2^{-i/2\sqrt{\log n}}$. So $T(n) = \sum_i Work_i$ is upper bounded by $f(n)\sum_i (2^{-1/2 sqrt \log n})^i = f(n)(1/1 - 2^{-1/2\sqrt{\log n}}) = f(n)O(\sqrt(\log n)$ using $2^{-1/x} \in 1 - \Theta(1/x)$ (which follows from the Taylor expansion of the function 2^x.) . So $T(n) \in O(f(n)\sqrt{\log n})$. On the other hand, for $i < \sqrt{\log n}$, $2^{-i/\sqrt{\log n}} \geq 1/2$, so $Work_i \in \Omega(f(n))$ for all such i. Therefore, $T(n) \in \Omega(f(n)\sqrt{\log n}$.

Thus, instead of being exactly steady-state, or exactly top-heavy, the recursion "splits the difference" adding a factor of $\sqrt{\log n}$ rather than either a constant or a factor of $\log n$. This is an unusual situation, but can in principle occur. (So even if it doesn't, we need to PROVE it doesn't, and cannot invoke a general lemma.)

Time Analysis: Irregular cases In some cases, instead of predictible sizes, the size of sub-problems depends on the instances, or even is a random variable. Then we need to analyze the possible sizes of the sub-problems. Say that one instance of size n is broken into a sub-problems, say of sizes $n_1, n_2, ..n_a$ and there is an additive factor of n^k.

One case that we can give the same kind of analysis for is the top-heavy case. If $\sum_{j=1}^{j=a}(n_j)^k < rn^k$ for some constant $r < 1$, we still have an exponentially decreasing amount of work as we go down levels. This gives time $T(n) = O(n^k)$.

The bottom-heavy case is similar, but a little tricky. If the sum is increasing exponentially, then like before it is a bottom-heavy case. But determining the number of base case recursions can be tricky.

If the above sum always equals n^k, then we are in the steady-state case. However, it doesn't necessarily hold that the number of recursion levels is $\Theta(\log n)$; this will be true if you can bound all sub-problems by $n_i < n/b$ for some constant $b > 1$.

Time analysis: Multiple parameters Sometimes divide and conquer algorithms have two size parameters and only one is decreasing. That tends to give a recurrence such as: $T(n, m) = aT(n/b, m) + g(m)f(n)$. We can use a trick in this case: Solve the recurrence $T(n) = aT(n/b) + f(n)$. The claim is that $T(n, m) = g(m)T(n)$ solves the two-parameter recurrence above. The argument is simple algebra, and the same as the one that constants in front of the additive part don't affect the asymptotic time.

Improving efficiency Some hints for getting better efficiency in divide-and-conquer algorithms:

1. For top-heavy algorithms, concentrate on improving the "conquer" time. Can this be improved, say with better data structures?
2. For steady-state algorithms, concentrate on moving as much work as possible from INSIDE the recursion to OUTSIDE . Is there some pre-processing that's possible to make the recursion more efficient? In other words, are we doing work at each level that could be done only once?
3. For bottom-heavy algorithms, can we reduce the number of recursive calls? Can different sub-problems be combined in some

way? Can some computation show that one or more of the subproblems are irrelevant? Can we show that not all levels of the recursion will make the maximum number of recursive calls?