D&C has the following basic steps:

- *Divide* the problem into smaller, similar sub-problems.
- *Conquer* each sub-problem recursively.
- *Combine* the sub-problem results to yield a result for the full problem.

A D&C algorithm, usually stated recursively, may be appropriate if the following conditions obtain:

- The problem can be divided into a fixed number of sub-problems.
- Each sub-problem is smaller than the original, and all are of similar size.
- Sub-problems have the same structure as the original problem.
- Sub-problems can be solved independently.
- Sub-problems’ results can be combined cheaply.

**Example: Sorting**

We want to sort an array of \( n \) elements by sorting some smaller sub-arrays, and combining the results. One way to do this would be to sort \( n - 1 \) elements, then insert the remaining element (i.e. insertion sort). This not the best application of D&C because it divides the problem into uneven-sized subparts – D&C is effective by reducing the problem size by a multiplicative (e.g. \( n/2 \)) rather than additive (e.g. \( n - 1 \)) factor. Indeed, insertion sort ends up being \( O(n^2) \).

A better approach is to divide the array into two equal-sized sub-arrays, recursively sort them, and combine the results. In a bit more detail:

```plaintext
1 Mergesort(A[1..n])
2   if n > 1
3     A1 <- Mergesort(A[1..n/2])
4     A2 <- Mergesort(A[n/2+1..n])
5   return Merge(A1, A2)
```
else
    return A

Merge(A[1..n], B[1..m])

i <- 1
j <- 1
C <- { Array [1..n+m] }
for k = 1 .. n + m
    if i = n
        C[k] <- B[j]
        j <- j + 1
    else if j = m
        C[k] <- A[i]
        i <- i + 1
    else if A[i] < B[j]
        C[k] <- A[i]
        i <- i + 1
    else
        C[k] <- B[j]
        j <- j + 1
return C

Here the divide step happens on lines 3-4, while the combine step happens on lines 5 and 9-26.

Running Time

Line 5 takes $O(n)$. To find the total time required, we need to state it as a recurrence

$$
T(n) = 2T(n/2) + O(n)
$$

$$
T(1) = O(1)
$$

We can think of all the calls as forming a tree where each node has two children, each of which is half the size of its parents, and has a cost proportional to its size. Since it bottoms out at $n = 1$, we know this tree has height $\log n$. This information can also be represented in a table:

<table>
<thead>
<tr>
<th>depth</th>
<th>sub-problems</th>
<th>size</th>
<th>cost per node</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>$cn$</td>
<td>$cn$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>n/2</td>
<td>$cn/2$</td>
<td>$cn$</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>n/4</td>
<td>$cn/4$</td>
<td>$cn$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>$2^i$</td>
<td>n/2^i</td>
<td>$cn/2^i$</td>
<td>$cn$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\log n$</td>
<td>n</td>
<td>1</td>
<td>$c$</td>
<td>$cn$</td>
</tr>
</tbody>
</table>

TOTAL

$cn \log n$
Implementation

While the recursive version is easy to analyze, an iterative implementation can be more efficient. Mergesort lends itself naturally to the following bottom-up iterative implementation:

1. \texttt{Mergesort(A[1..n])}
2. \texttt{j <- 1}
3. \texttt{while j < n}
   4. \texttt{for i = 1 to n-2*j by 2*j}
      5. \texttt{Merge_inplace(A[i .. i+j], A[i+j+1 .. i+2*j])}
   6. \texttt{j <- j * 2}

Since this iterative version does all the same work as the recursive version, its asymptotic running time is the same, while its constant factors will be smaller (Exercise: show the correspondence between merges in the iterative and recursive versions of Mergesort).

Example: All Pairs Binary Tree Distances

Input: A complete binary tree \( T \) with \( n \) nodes and distances on the edges.

Output: The total distance between every pair of nodes \((i, j)\).

The naive approach of considering every pair of nodes separately takes time \( O(n^2 \log n) \), because there are \( n^2 \) pairs of nodes, and a path between them has length \( \log n \). Since the the size of our output is \( n^2 \), we know the algorithm must be at least \( O(n^2) \). However, since the naive approach repeats a lot of work, we suspect that we can do better, and since we are dealing with a recursive problem on trees, D&C may be appropriate.

The D&C algorithm first (recursively) computes distances in the subtrees rooted at each of a node’s children, then updates distances to reflect the paths to the parent node, and between nodes in opposite child subtrees.

1. \texttt{M <- \{ Array [1..n, 1..n] of distances \}}
2. \texttt{M <- Infiniti}
3. \texttt{Alldist(t)}
   4. \texttt{if !has_children(t)}
      5. \texttt{return}
   6. \texttt{Alldist(left(t))}
   7. \texttt{Alldist(right(t))}
   8. \texttt{M[left(t),t] <- dist(t,left(t))}
   9. \texttt{M[right(t),t] <- dist(t,right(t))}
10. \texttt{for c = \{ descendants(left(t)) \}}
   11. \texttt{M[c,t] <- M[c,left(t)] + dist(t,left(t))}
12. \texttt{for c = \{ descendants(right(t)) \}}
   13. \texttt{M[c,t] <- M[c,right(t)] + dist(t,right(t))}
14. \texttt{for c = \{ descendants(left(t)) \}}
for d = \{ \text{descendants(right(t))} \}
M[c,d] = M[c,t] + M[t,d]

**Running Time**

Lines 1-2 each take $O(n^2)$. Each subtree has size $n/2$, so the loops on lines 10 and 12 repeat $n/2$ times, while the nested loops on lines 14-16 run $(n/2)^2$ times. The operations in these loops take $O(1)$, so their total cost is $O(n^2)$. The two recursive calls on lines 6-7 take $T(n/2)$, so the recurrence relation is

$$
T(n) = 2T(n/2) + O(n^2)
$$
$$
T(1) = O(1)
$$

This looks like $O(n^2 \log n)$, which is disappointing given how much more complicated our algorithm has become. But let’s write down the table:

<table>
<thead>
<tr>
<th>depth</th>
<th>sub-problems</th>
<th>size</th>
<th>cost per node</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$c n^2$</td>
<td>$c n^2$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
<td>$c(n/2)^2$</td>
<td>$cn^2/2$</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$n/4$</td>
<td>$c(n/4)^2$</td>
<td>$cn^2/4$</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>$2^i$</td>
<td>$2^i$</td>
<td>$c(n/2^i)^2$</td>
<td>$cn^2/2^i$</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$2^n$</td>
<td>1</td>
<td>$c$</td>
<td>$cn^2/2^{\log n}$</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>$cn^2(1/2 + 1/4 + \cdots + 1/n)$</td>
<td>$2cn^2 = O(n^2)$</td>
</tr>
</tbody>
</table>

Happily, the total time is actually $O(n^2)$. Note that the table illustrates that the tree is top-heavy, i.e. that almost all the work is done at the top level.