Divide and Conquer

K. Subramani
Department of Computer Science and Electrical Engineering,
West Virginia University,
Morgantown, WV
ksmani@csee.wvu.edu

1 Strategy

Function DIVIDE-AND-CONQUER (A, n)

```
Usually the input to the algorithm is an array A and a size n. This is your input problem P.
if (n ≤ c) then
   The problem is simple to solve, because it has a small size.
   return.
end if
Break up the problem into a number of sub-problems say k. Let us call the sub-problems P_1, P_2, ..., P_k. Let
their corresponding sizes be n_1, n_2, ..., n_k. This is the Divide phase!
for ( i=1 to k ) do
   Recursively solve problem P_i. Let C_i be the solution obtained by solving P_i. This is the Conquer phase!
end for
COMBINE the C_i to form the solution for the input problem P.
```

Algorithm 1.1: The Divide and Conquer Approach

2 Analysis

The key point is that solving sub-problems is easier than solving the larger input problem. For example, sorting 2 numbers is easier than sorting 100 numbers. Another point is that the sub-problems are solved recursively i.e. the same algorithm which solves the initial problem also solves the sub-problem. The break-up continues till some point when a sub-problem size drops below c; at this point the problem can be solved using some simple technique. *Finally, the cost of the strategy depends upon the cost of the dividing strategy, the cost of the conquering and the cost of the combining techniques.*

Let $T(n)$ denote the cost of the problem when it is of size n. Then, we have,

$$T(n) = T(n_1) + T(n_2) + \ldots T(n_k) + C(P_1, P_2, \ldots, P_k) \tag{1}$$

where C is the cost of combining solutions.

3 Examples

3.1 Merge-Sort

Function MERGE-SORT (A, p, q)

The problem P is to sort the elements in $A[p..q]$. Initially, i.e. when this function is called from your main program, $p = 1$ and $q = n$.
if $(p \geq q)$ **then**
  There is at most one element in the array; so there is no need to sort.
  return.
end if
Here $p < q$.
$mid = \frac{p+q}{2}$.
MERGE-SORT(A, p, mid)
MERGE-SORT($A, mid + 1, q$)
MERGE(A, p, q, mid)

Algorithm 3.1: Merge-Sort

The MERGE procedure is described in the text. The key point is merging two sorted arrays of sizes a and b takes time $a+b$. In our case, we have two sorted arrays of size $\frac{n}{2}$ and hence time taken for the combining strategy i.e. MERGE is n [1]. Hence, we have,

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + n \\ &= 2.T(\frac{n}{2}) + n \\ &= 2.[2.T(\frac{n}{4}) + \frac{n}{2}] + n \text{ (} \textit{Reapplying the recurrence definition} \text{)} \\ &= 4.T(\frac{n}{4}) + 2.n \\ &= 8.T(\frac{n}{8}) + 3.n \\ &= 2^k.T(\frac{n}{2^k}) + k.n, \end{aligned} \tag{2}$$

where

$$2^k = n \Rightarrow k = \log_2 n$$

Hence,

$$T(n) = n.T(1) + \log n.n$$

In our program, $T(1) = 1$, since we have to check that $p \geq q$. Thus,

$$T(n) = n.1 + n.\log n \leq 2.n \log n = O(n.\log n)$$

3.2 Quick-Sort

Here the Divide straegy is non-trivial. Indeed, the core of the algorithm is how to effect the partition or division. The PARTITION procedure is described below.

Unfortunately, the partition procedure cannot guarantee that the division will be into equal parts. Suppose for instance, that the input array A is sorted *in reverse order.* Then PARTITION will return two arrays one of size 1 and the other of size $n - 1$. Hence the worst-case complexity is given by:

$$T(n) = T(n-1) + n \tag{3}$$

which gives $T(n) = O(n^2)$.

[1] The dividing strategy is trivial i.e. it takes constant time

Function QUICK-SORT (A, p, q)

The problem P is to sort the elements in $A[p..q]$. Initially, i.e. when this function is called from your main program, $p = 1$ and $q = n$.
if $(p \geq q)$ **then**
    There is at most one element in the array; so there is no need to sort.
    return.
end if
Here $p < q$.
$j =$ PARTITION(A, p, r)
QUICK-SORT(A, p, j)
QUICK-SORT$(A, j+1, q)$

Algorithm 3.2: Quick-Sort

Function PARTITION (A, p, q)

$x = A[p]; \ i = p - 1; \ j = q + 1$
The problem is to parition the elements in $A[p..q]$ around $A[p]$, so that all elements less than or equal to $A[p]$ fall in the left portion and all other elements fall in the right portion.
while (**true**) **do**
    repeat
        $j \leftarrow j - 1;$
    until $A[j] \leq x$.
    repeat
        $i \leftarrow i + 1;$
    until $A[i] \geq x$.
    if $(i < j)$ **then**
        exchange$(A[i], A[j])$
    else
        return(j)
    end if
end while

Algorithm 3.3: Partition