Divide and Conquer

Designing an efficient algorithm for solving a problem requires careful analysis of the problem at hand. Luckily, there are a few meta-techniques that we can adopt to deduce suitable algorithms and one such approach is the "divide and conquer" approach.

The divide-and-conquer approach involves partitioning a problem into *disjoint* subproblems and then tackling the subproblems separately. The solutions to the subproblems may then need to be combined to obtain the solution to the original problem.

In this note, we will explore two divide-and-conquer algorithms, Binary Search and Quicksort.

1 Binary Search

One of the most elementary applications of the divide-and-conquer method is to the problem of finding an element in a sorted list. If we have to find an element x in a sorted list A containing n elements, we can resort to the Binary Search algorithm (Algorithm 1). The simplest version of this algorithm merely returns *true* if the element x is present in the list else it returns *false*. The algorithm performs a check to see if the midpoint of the list is the desired element or not. If it is not then, recursively, the search progress to the half of the list where the element is likely to exist.

1.1 Proof of correctness

We will first establish the correctness of binary search. The proof relies on mathematical induction.

The base case is the empty list, which is specified by one of the three cases: $i < 1$ or $j > n$ or $i > j$ because i is the start index for the search, j is the

Algorithm 1 Binary Search

```
bool binarysearch(A[i, j], x)
if (i < 1) ∨ (j > n) ∨ (i > j) then
  return false
end if
mid ← ⌊(i + j)/2⌋
if A[mid] = x then
  return true
else
  if A[mid] < x then
    binarysearch(A[mid + 1, j], x)
  else
    binarysearch(A[i, mid − 1], x)
  end if
end if
```

end index for the search, and the list has n elements. In this case it is clear that the algorithm correctly returns *false*.

We assume, as the induction hypothesis, that the algorithm works correctly for lists of length $n = 1, 2, \ldots, k$ and we need to show that it will work correctly for a list of length $k+1$.

For a list of length $k+1$, we will begin with a call to the search the list A for x between indices 1 and $k+1$. This yields two cases. (Case 1.) If $A[mid]$ contains x then the algorithm correctly returns *true*. (Case 2.) If $A[mid]$ does not contain x then, using the fact that the list is sorted, in a non-decreasing order in the example, we only need to check the left or right half of the list depending on the inequality $A[mid] < x$. Then, by the inductive hypothesis, the recursive call returns the correct answer because the call is to a list of size smaller than $k+1$. ■

1.2 Runtime complexity

Let us suppose that $T(n)$ represents the running time of binary search on a list of length n. We can compute *mid* and compare $A[mid]$ with x in constant time. Based on the result of the comparison, we may have to search a list of

length $n/2$. If $n < 1$ then the algorithm runs in constant time. Therefore,

$$\begin{aligned} T(n) &\leq \Theta(1) + T(n/2), \\ T(1) &= \Theta(1). \end{aligned}$$

We can unroll the recurrence for $T(n)$ as

$$T(n) \leq \Theta(1) + T(n/2) \leq 2\Theta(1) + T(n/4) \leq 3\Theta(1) + T(n/8) \leq \ldots$$

The termination of the recurrence is when $n = 1$ and therefore the maximum depth of the recursion is $O(\lg n)$. This yields a $O(\lg n)$ time complexity for binary search.

2 Quicksort

Quicksort is a divide-and-conquer sorting algorithm. When sorting a list A, Quicksort works as follows:

- An element is selected to be the pivot element, and the list A is rearranged such that all elements that are smaller than the pivot appear before the pivot and all elements larger than the pivot appear after the pivot.
- The partition containing elements less than the pivot and the partition containing elements greater than the pivot are sorted recursively.

A version of Quicksort uses additional space (Algorithm 2) and it is possible to implement Quicksort such that it uses in-place sorting.

It is possible to show the correctness of Quicksort using a simple inductive proof. Understanding the number of comparisons performed by Quicksort is more interesting. The runtime of Quicksort, $T(n)$, depends on the number of comparisons performed, and the number of comparisons depends on the choice of the pivot.

Generally, if the correct position for the pivot element is k in a list of length n then

$$T(n) = \Theta(n) + T(k-1) + T(n-k),$$

Algorithm 2 Quicksort

```
function quicksort(A)
if length(A) ≤ 1 then
    Comment: The list is sorted by default
    return (A)
end if
select and remove a pivot value p from A
create empty lists Less and Greater
for each x in A do
    if x ≤ p then
        append x to Less
    else
        append x to Greater
    end if
end for
return concatenate(quicksort(Less), p, quicksort(Greater))
```

because of the two recursive calls and the partitioning about the pivot which takes $\Theta(n)$ time (Algorithm 2).

If we choose the pivot carefully (to be the median of the list) then the Quicksort recurrence becomes

$$T(n) = \Theta(n) + 2T(n/2).^{1}$$

To find a bound on $T(n)$, with the median element being the pivot, we can use a recursion tree (Figure 1). We see that at each level of the tree there is a total of $\Theta(n)$ work that needs to be performed for the partitioning. (Concatenation can be performed in constant time depending on how the lists are implemented.) The height of tree is no more than $1 + \lfloor \lg n \rfloor$. This gives us a $\Theta(n \lg n)$ bound on the running time of Quicksort. In this analysis, we have implicitly assumed that we can find the median quickly (in $\Theta(n)$ time). Can we do so?

[1] The rounding does not matter too much.

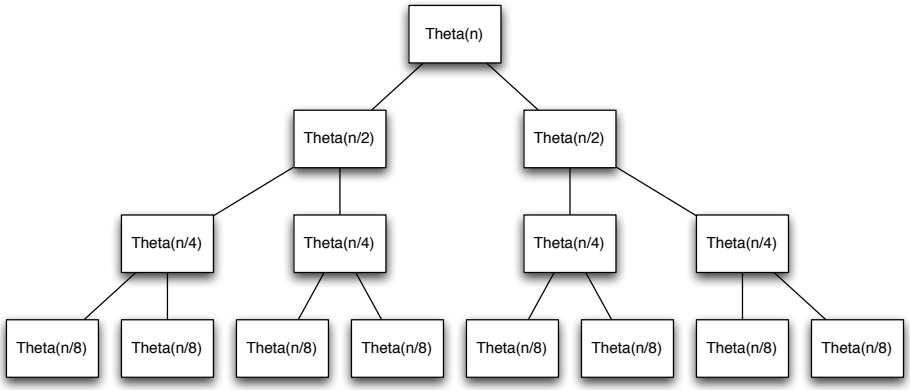

Figure 1: Recursion tree for Quicksort

3 Points to ponder

We have seen two simple divide-and-conquer algorithms in this note. We see that induction is a convenient technique to prove the correctness of these algorithms because of the recursive nature of the algorithms. We have seen, briefly, the use of recurrence equations to obtain bounds on the running times and we shall study more general techniques to obtain such bounds. In the two algorithms we have seen, combining the results from the subproblems is trivial. We shall consider situations when the partitioning into sub-problems as well as the combination of sub-solutions requires more sophistication.