1 Divide & Conquer

One strategy for designing efficient algorithms is the "divide and conquer" approach, which is also called, more simply, a recursive approach. The analysis of recursive algorithms often produces mathematical equations called *recurrence relations*. It is the analysis of these equations that produces the bound on the algorithm running time.

The divide and conquer strategy has three basic parts. For a given problem of size n,

1. **divide**: break a problem instance into several smaller instances of the same problem
2. **conquer**: if a smaller instance is trivial, solve it directly; otherwise, divide again
3. **combine**: combine the results of smaller instances together into a solution to a larger instance

That is, we split a given problem into several smaller instances (usually 2, but sometimes more depending on the problem structure), which are easier to solve, and then combine those smaller solutions together into a solution for the original problem. The goal is to keep dividing until we get the problem down to a small enough size that we can solve it quickly (or trivially). Fundamentally, divide and conquer is a recursive approach, and most divide and conquer algorithms have the following structure:

```
function fun(n) {
   if n==trivial {
      solve and return
   } else {
      partA = fun(n')
      partB = fun(n-n')
      AB    = combine(A,B)
      return AB
   }
}
```

The recursive structure of divide and conquer algorithms makes it useful to model their asymptotic running time $T(n)$ using recurrence relations, which often have the general form

$$T(n) = a\,T(g[n]) + f(n) \ , \tag{1}$$

where a is a constant denoting the number of subproblems we break a given instance into, $g[n]$ is a function of n that describes the size of the subproblems, and $f(n)$ is the time required to combine the smaller results / divide the problem into the smaller versions. There are several strategies for solving recurrence relations. We'll cover in this unit: the "unrolling" method and the master

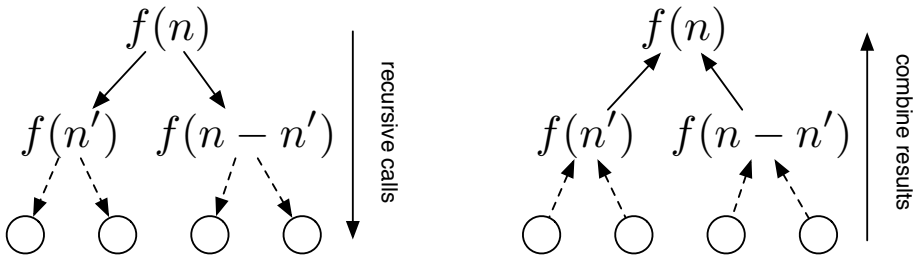

Figure 1: Schematic of the divide & conquer strategy, in which we recursively divide a problem of size n into subproblems of size n' and $n - n'$, until a trivial case is encountered (left side). The results of each pair of subproblems are combined into the solution for the larger problem. The computation of any divide & conquer algorithms can thus be viewed as a tree.

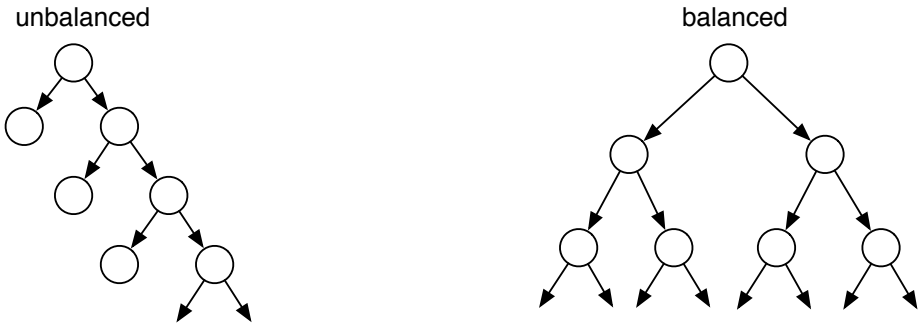

Figure 2: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$. As we go through the QuickSort analysis below, keep these pictures in mind, as they will help you understand why QuickSort is a fast algorithm.

method; other methods include annihilators, changing variables, characteristic polynomials and recurrence trees. You can read about many of these in the textbook (Chapter 4.2).

Consider the case of $a = 2$. Figure 1 shows a schematic of the way a generic divide and conquer algorithm works. The algorithm effectively builds a computation or recursion tree, in which an internal node represents a specific non-trivial subproblem. Trivial or base cases are located at the leaves. As the function explores the tree, it uses a *stack* data structure (CLRS Chapter 10.1) to store the previous, not-yet-completed problems, to which the computer will return once it has completed the subproblems rooted at a given internal node. The path of the calculation through the recursion tree is a depth-first exploration.

The division of a problem of size n into subproblems of sizes $n - n'$ and n' determines the depth the tree, which determines how many times we incur the $f(n)$ cost. The sum of these costs is the running time. Consider the cases of $n' = 1$ and $n' = n/2$ (see Figure 2). In the first case, each time we recurse, we carve off a trivial case from the current size, and this produces highly unbalanced recursion trees, with depth n. In the second case, we divide the problem in half each time, and the depth of the tree is $\lg n$. If $f(n) = \omega(1)$, then the total cost is different between the two cases; in particular, the more balanced case is lower cost.

2 Quicksort

Quicksort is a classic divide and conquer algorithm, which uses a comparison-based approach for sorting a list of n numbers. In the naïve version, the worst case is $\Theta(n^2)$ but its expected (average) is $O(n \log n)$. This is asymptotically faster than algorithms like Bubble Sort[1] and Insertion Sort,[2] whose average and worst cases are both $\Theta(n^2)$. The deterministic version of Quicksort also exhibits $O(n^2)$ worst case behavior, which is not particularly fast.

However, Quicksort's average case is $O(n \log n)$, which is provably the asymptotic lower bound on comparison-based sorting algorithms. By using randomness in a clever way, we can trick Quicksort into behaving as if every input is an average input, and thus producing very fast sorting. This is just as fast as another divide and conquer sorting algorithm called Mergesort.[3]

This randomized version of Quicksort is generally considered the best sorting algorithm for large inputs. If implemented correctly, Quicksort is also an *in place* sorting algorithm, meaning that it requires only $O(1)$ "scratch" space in addition to the space required for the input itself.[4]

Here are the divide, conquer and combine steps of the Quicksort algorithm:

[1] Bubble Sort. Let A be an array with n elements. While any pair $A[i]$ and $A[i+1]$ are out of order, let $i = 1$ to $n - 1$ and for each value of i, if $A[i]$ and $A[i+1]$ are out of order then swap them.

[2] Insertion Sort. Let A be an array with n elements. For $i = 2$ to n, first set $j = i$. Then, while $j > 1$ and $A[j-1]$, $A[j]$ are out of order, swap them and decrement j.

[3] Mergesort. Let A be an array with n elements. If $n = 1$, A is already sorted. Otherwise, divide A into two equal sized subarrays and call Mergesort on each. Then step through the resulting two arrays, left-to-right, and merge them so that A is now sorted and return A.

[4] Many implementations of Quicksort are not in-place algorithms because they copy intermediate results into an array the same size as the input; similarly, "safe recursion," which passes variables by copy rather than by reference, does not yield an in-place algorithm.

1. **divide**: pick some element $A[q]$ of the array A and partition A into two arrays A_1 and A_2 such that every element in A_1 is $\leq A[q]$ and every element in A_2 is $> A[q]$. We call the element $A[q]$ the *pivot* element.
2. **conquer**: Quicksort A_1 and Quicksort A_2
3. **combine**: return A_1 concatenated with $A[1]$ concatenated with A_2, which is now the sorted version of A.

The trivial or base case for Quicksort can either be sorting an array of length 1, which requires no work, or sorting an array of length 2, which at most requires one comparison, three assignment operations and one temporary variable of constant size. Thus, the base case takes $\Theta(1)$ time and $\Theta(1)$ scratch space.

2.1 Pseudocode

Here is pseudocode for the main Quicksort procedure, which takes an array A and array bounds p, r as inputs.

```
// Precondition:  A is the array to be sorted, p>=1;
//                r is <= the size of A
// Postcondition: A[p..r] is in sorted order

Quicksort(A,p,r) {
   if (p<r){
      q = Partition(A,p,r)
      Quicksort(A,p,q-1)
      Quicksort(A,q+1,r)
}}
```

This procedure calls another function `Partition` before it hits either recursive call. This implies that `Partition` must do something such that no additional work is required to stitch together the solutions of the subproblems. `Partition` takes the same types of inputs as Quicksort:

```
//Precondition:  A[p..r] is the array to be partitioned, p>=1 and r<= size of A,
//               A[r] is the pivot element
//Postcondition: Let A' be the array A after the function is run. Then A'[p..r]
//               contains the same elements as A[p..r]. Further, all elements in
//               A'[p..res-1] are <= A[r], A'[res] = A[r], and all elements in
//               A'[res+1..r] are > A[r]

Partition(A,p,r) {
   x = A[r]
   i = p-1
   for (j=p; j<=r-1;j++) {
      if A[j]<=x {
         i++
         exchange(A[i],A[j])
   }}
   exchange(A[i+1],A[r])
   return i+1
}
```

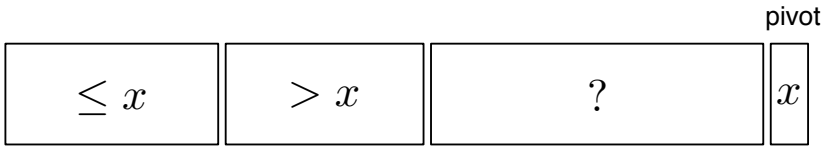

Figure 3: The intermediate state of the `Partition` function: values in the "$\leq x$" region have already been compared to the pivot x, found to be less than or equal in size and moved into this region; values in the "$> x$" region have been compared to x, found to be greater in size and moved into this region; values in the "?" region are each compared to x and then moved into either of the two other regions, depending on the result of the comparison.

2.2 Correctness

Whenever we design an algorithm, we must endeavor to prove that it is *correct*, i.e., it yields the proper output on *all* possible inputs, even the worst of them. In other words, a correct algorithm does not fail on any input. If there exists even one input for which an algorithm does not perform correctly, then the algorithm is not correct. Our task as algorithm designers is to produce correct algorithms; that is, we need to prove that our algorithm never fails.

To show that an algorithm is correct, we identify some mathematical structure in its behavior that allows us to show that the claimed performance holds on all possible inputs.

To prove that Quicksort is correct, we use the following insight: at each intermediate step of `Partition`, the array is composed of exactly 4 regions, with x being the pivot (see Figure 3):

- Region 1: values that are $\leq x$ (between locations p and i)
- Region 2: values that are $> x$ (between locations $i+1$ and $j-1$)
- Region 3: unprocessed values (between locations j and $r-1$)
- Region 4: the value x (location r)

Crucially, throughout the execution of `Partition`, regions 1 and 2 are growing, while region 3 is shrinking. At the end of the loop inside `Partition`, region 3 is empty and all values except the pivot are in regions 1 and 2; `Partition` then moves the pivot into its correct and final place.

To prove the correctness of Quicksort, we will use what's called a *loop invariant*, which is a set of properties that are true at the beginning of each cycle through the `for` loop, for any location k. For QuickSort, here are the loop invariants:

1. If $p \leq k \leq i$ then $A[k] \leq x$
2. If $i+1 \leq k \leq j-1$ then $A[k] > x$
3. If $k = r$ then $A[k] = x$

Verify for yourself (as an at-home exercise) that (i) this invariant holds before the loop begins (initialization), (ii) if the invariant holds at the $(i-1)$th iteration, that it will hold after the ith iteration (maintenance), and (iii) show that if the invariant holds when the loop exists, that the array will be successfully partitioned (termination). This proves the correctness of `Partition`.

The correctness of Quicksort follows by observing that after `Partition`, we have correctly divided the larger problem into two subproblems—values less than the pivot and values greater than the pivot—both of which we solve by calling `Quicksort` on each. After the first call to `Quicksort` returns, the subarray `A[p..q-1]` is sorted; after the second call returns, the subarray `A[q1..r]`+ is sorted. Because `Partition` is correct, the values in the first subarray are all less than $A[q]$, and the values in the second subarray are all greater than $A[q]$. Thus, the subarray $A[p..r]$ is in sorted order, and Quicksort is correct.

2.3 An example

To show concretely how Quicksort works, consider the array $A = [2, 6, 4, 1, 5, 3]$. The following table shows the execution of Quicksort on this input. The pivot for each invocation of `Partition` is in **bold face**.

procedure	arguments	input	output	global array
first partition	$x = 3 \;\; p = 0 \;\; r = 5$	$A = [2, 6, 4, 1, 5, \mathbf{3}]$	$[2, 1 \mid \mathbf{3} \mid 6, 5, 4]$	$[2, 1, 3, 6, 5, 4]$
L QS, partition	$x = 1 \;\; p = 0 \;\; r = 1$	$A = [2, \mathbf{1}]$	$[\mathbf{1} \mid 2]$	$[1, 2, 3, 6, 5, 4]$
R QS, partition	$x = 4 \;\; p = 3 \;\; r = 5$	$A = [6, 5, \mathbf{4}]$	$[\mathbf{4} \mid 5, 6]$	$[1, 2, 3, 4, 5, 6]$
L QS, partition	do nothing			$[1, 2, 3, 4, 5, 6]$
R QS, partition	$x = 6 \;\; p = 4 \;\; r = 5$	$A = [5, \mathbf{6}]$	$[5 \mid \mathbf{6}]$	$[1, 2, 3, 4, 5, 6]$

3 On your own

1. Read Chapters 4 and 7

A brief aside about proofs by induction

Induction is a simple and powerful technique for proving mathematical claims, and is one of the most common tools in algorithm analysis, precisely because of these properties. When we invoke this approach to proving a claim, we must state exactly what variable or property we are using induction *on*.

Consider the example from Lecture 1, the sum of the first n positive integers, $S(n) = 1+2+3+\cdots+n$. We claim that a closed form for this sum is

$$S(n) = \frac{n(n+1)}{2} \ . \tag{2}$$

We will mathematically prove, via induction on n, that $n(n+1)/2$ is the correct closed-form solution.[5] A proof by induction has two parts:

- **Base case**: in which we verify that $S(n)$ is correct for the smallest possible value, in this case $n = 1$. (If the claim is about some recurrence relation, the base cases will often be given to you, e.g., $T(0) = c_1$ and $T(1) = c_2$, for two base cases where c_1 and c_2 are constants.) That is,

$$S(1) = \frac{1(1+1)}{2} = 1 \ .$$

- **Inductive step**: in which we (i) assume that $S(n)$ holds for an arbitrary input of size n and then (ii) prove that it also holds for $n+1$. In practice, this amounts to taking Eq. (2), or its equivalent in your problem of choice, and replacing each instance of n with an instance of $n+1$ and the simplifying:

$$\begin{aligned} S(n+1) &= \frac{(n+1)((n+1)+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \\ &= \frac{n^2+3n+2}{2} \\ &= \frac{n^2+n+2n+2}{2} \\ &= \frac{n(n+1)}{2} + \frac{2n+2}{2} \\ &= S(n) + (n+1) \ , \end{aligned}$$

[5] Here, there is only one possible choice, but for more complex algorithms, there will be several choices. In those cases, our job is to choose correctly, so as to make the induction straightforward. A hint that you have chosen incorrectly is that the induction is hard to push through.

where in the final step we have used the definition of $S(n)$ to reduce a more complicated expression into a simple one completes the inductive step. (In some problems, there will be cases to consider, e.g., $n \geq c$ for c constant and $n < c$, and in that situation, the inductive step must be applied to each.)

In this class, we aim to present more formal proofs. Thus, rather than the more wordy, thinking-out-loud version above, here is a compact, formal proof, which states the claim precisely, states the assumptions clearly, and shows exactly what is necessary to understand the proof.

Theorem: The sum of the first n positive integers is $S(n) = n(n+1)/2$.

Proof: Let n be an arbitrary positive integer, and assume inductively that $S(n) = n(n+1)/2$. The base case $n = 1$ is $S(1) = 1(1+1)/2 = 1$, and for any $n \geq 1$ we have

$$\begin{aligned} S(n+1) &= \frac{(n+1)((n+1)+1)}{2} && \big[\text{definition of } S(n+1)\big] \\ &= \frac{n(n+1)}{2} + \frac{2n+2}{2} && \big[\text{algebra}\big] \\ &= S(n) + (n+1) && \big[\text{inductive hypothesis with } S(n)\big] \ , \end{aligned}$$

□

If you would like to see more examples of proofs by induction or get a deeper mathematical explanation of why and how they work, see the excellent notes by Jeff Erickson at UIUC:
`http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/98-induction.pdf`