1 Divide & Conquer

"Divide and conquer" is one strategy for solving problems. It has three basic parts:

1. **divide**: break a problem instance into several smaller instances of the same problem
2. **conquer**: solve these smaller cases (return to divide step)
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. 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 performance using to recurrence relations. Such recurrence relations often have the form

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

where a is some constant denoting the number of subproblems we break a given instance into, $g(n)$ is some function of n that describes the size of the subproblems, $f(n)$ is the time required to solve the trivial or base case. There are several strategies for solving recurrence relations. We'll cover two today: the "unrolling" method and the master 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).

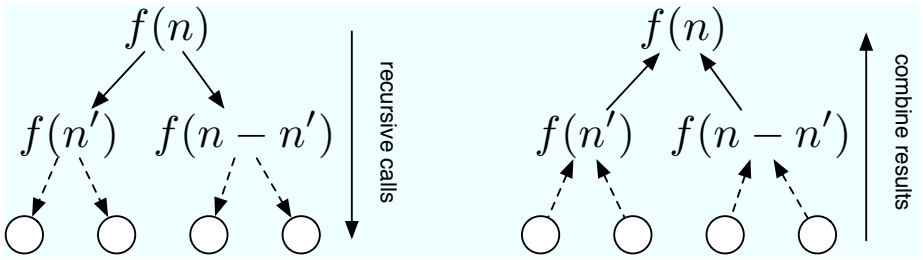

Figure 1: Schematic of the divide & conquer strategy, in which a large problem n is divided into smaller problems (of size n' and $n-n'$), which themselves may be further subdivided, until a trivial case is encountered. Then, the results of the subproblems are combined to be the result of the larger problem. The computation of any divide & conquer algorithms can thus be viewed as a tree.

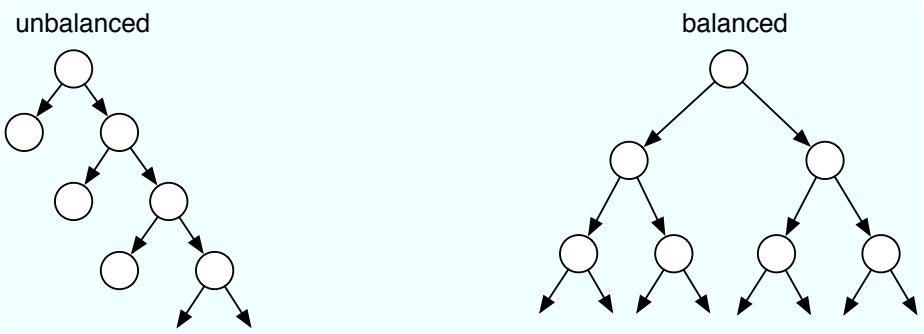

Figure 2: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$. As we go through the QuickSort analysis below, keep these pictures in mind, as they will help you understand why QuickSort is a fast algorithm.

2 Quicksort

Quicksort is a classic divide and conquer algorithm, which uses a comparison-based approach to 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 faster than more naïve sorting algorithms like Bubble Sort or Insertion Sort, whose average and worst cases are both $O(n^2)$. The randomized version of Quicksort that we'll encounter below is widely considered the best sorting algorithm for large inputs. All versions of Quicksort are *in place* sorting algorithms, meaning that they require no additional space. (Another divide-and-conquer sorting algorithm is Mergesort, which takes $O(n \log n)$ time.)

Here are the divide, conquer and combine steps of the Quicksort 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 requires one comparison and possibly one swap.

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 uses another function `Partition` to do the division step; this procedure 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
}
```

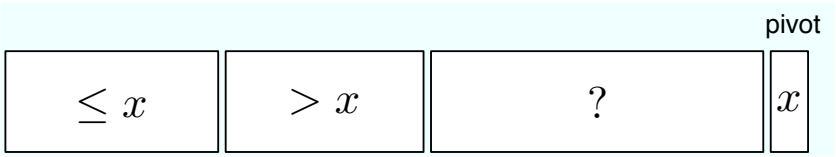

Figure 3: The intermediate state of the `Partition` function: values in the "$\leq x$" region have already been compared to the pivot x, found to be less than or equal in size and moved into this region; values in the "$> x$" region have been compared to x, found to be greater in size and moved into 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

For an algorithm to be *correct*, it must yield the proper output on *all* possible inputs (even the worst of them). To show that an algorithm is correct, we should identify some mathematical structure in its behavior that allows us to show that the claimed performance holds on all possible inputs.

The basic idea: 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)

Regions 1 and 2 are growing, while region 3 is shrinking. When the algorithm completes, region 3 is empty and all values are in regions 1 and 2.

One way to prove the correctness of this algorithm is to define a *loop invariant*, which is a set of properties that are true at the beginning of each iteration of 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).

2.3 An example

Consider the array $A = [2, 6, 4, 1, 5, 3]$. The following table shows what Quicksort does. 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	$x = 6 \;\; p = 4 \;\; r = 5$	$A = [5, \mathbf{6}]$	$[5 \mid \mathbf{6}]$	$[1, 2, 3, 4, 5, 6]$

2.4 Analysis

The `Partition` function examines each element in the subarray passed into it and thus runs in linear time. This in turn means that the total running time of Quicksort will depend on whether the partitioning (the recursive step) is balanced or not, and this depends on which element in A is used as the pivot. To see why, we'll examine the worst- and best-case performances. (We'll do average case next time.)

2.4.1 Worst case

The worst possible performance must come when the tree describing QuickSort's recursive steps is maximally *unbalanced* (see Figure 2).

The most unbalanced the partitions can be is to repeatedly divide A such that one piece has a constant number of elements (at worst 0) and the other piece has all the rest. Thus, the worst case

recurrence relation looks like

$$T(n) = T(n-1) + \Theta(n) \ . \tag{2}$$

This kind of recurrence relation, in which $g(n) = n - b$ for some constant b, is also characteristic of "tail recursion" algorithms, which are logically equivalent to `for` loops. It can easily be solved using the "unrolling" method:

$$\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
T(n) &= T(n-2) + \Theta(n-1) + \Theta(n) \\
T(n) &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&\vdots \\
T(n) &= \Theta(1) + \cdots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
T(n) &= \sum_{i=1}^{n} \Theta(i) \\
T(n) &= \Theta(n^2) \ .
\end{aligned}$$

2.4.2 Best case, and the Master Theorem

The best case happens when the partitions are perfectly balanced, that is, when we choose a pivot exactly in the middle (the *median*) of the list. In this case, the recurrence relation is

$$T(n) = 2T(n/2) + \Theta(n) \ . \tag{3}$$

That is, because we divide the list into 2 equal halves, the recursive call produces 2 subproblems each half the size as our current problem. To solve this recurrence relation, we will use the *master method* (Chapter 4.3), which can be applied to any recurrence relation in which $g(n) = n/b$ for some constant b.

Master theorem
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence,

$$T(n) = a\,T(n/b) + f(n) \ ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$

3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. □

The master method does not require any thought — all of that was put into proving it — so it does not require much algebra.

Before applying it, let's consider what it's doing. Each of the three cases compares the asymptotic form of $f(n)$ with $n^{\log_b a}$; whichever of these functions is larger dominates the running time.[1] In Case 1, $n^{\log_b a}$ is larger so the running time is $\Theta(n^{\log_b a})$; in Case 2, the functions are are the same size, so we multiply by a $\log n$ factor and the solution is $\Theta(f(n)\log n)$; in Case 3, $f(n)$ is larger, so the solution is $\Theta(f(n))$.[2] [3]

Applying the master method to Eq. (3) is straightforward. Note that $a = b = 2$ and $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$ which is Case 2, in which $n^{\log_b a}$ and $f(n)$ are the same size, so $T(n) = \Theta(n \log n)$.

(As an at-home exercise, try using the "unrolling" method to solve Eq. (3).)

3 Next time

1. We'll analyze the average case of Quicksort
2. Read Chapters 4 and 7 of CLRS

[1] In fact, the function much be *polynomially* smaller, which is why there's a factor of n^{ϵ} in the theorem.

[2] Similar to Case 1, there is a *polynomially* larger condition here, plus a "regularity" condition in which $a\,f(n/b) \leq c, f(n)$, but most polynomially bounded functions we encounter will also satisfy the regularity condition.

[3] Another detail is that the Master Theorem does not actually cover all possible cases. There are classes of functions that are not covered by the three cases; if this is true, then the Master Theorem cannot be applied.