1 Divide & Conquer Algorithms

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

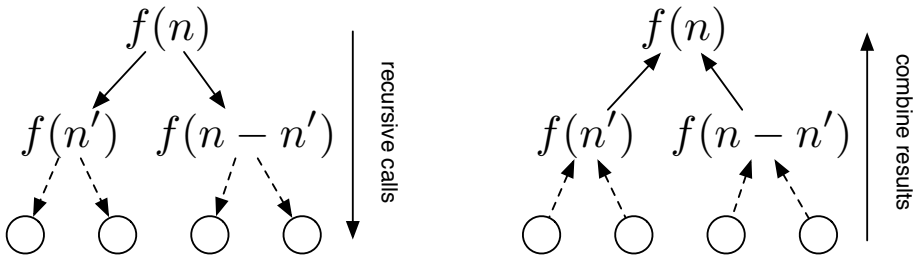

Figure 1: Schematic of the divide & conquer strategy, in which we recursively divide a problem of size n into subproblems of size n' and $n - n'$, until a trivial case is encountered (left side). The results of each pair of subproblems are combined into the solution for the larger problem. The computation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

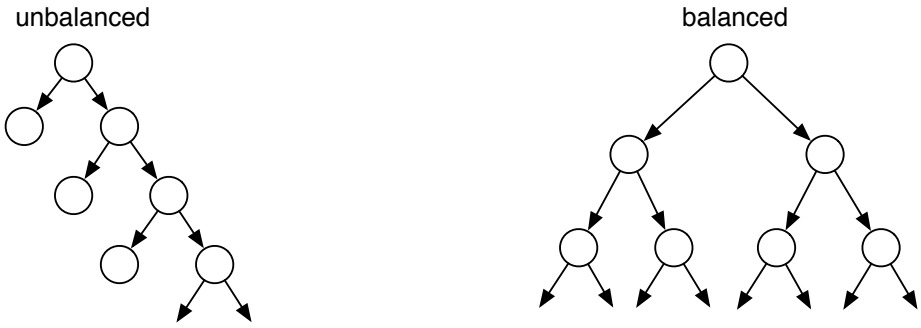

Figure 2: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$. As we go through the QuickSort analysis below, keep these pictures in mind, as they will help you understand why QuickSort is a fast algorithm.

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 (deterministic)

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]

Below are the divide, conquer and combine steps of the Quicksort algorithm. This high-level description of the steps is correct for both deterministic and randomized versions of the 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 an 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 base case for Quicksort is either sorting an array of length 1, which requires no work, or sorting an array of length 2, which requires at most one comparison, three assignment operations and one temporary variable of constant size. Thus, a 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 makes either of the two recursive calls. This implies that `Partition` must do something such that no additional work is required to stitch together the solutions of the subproblems, because after the final recursive call returns, the top-level call to Quicksort immediately returns the contents of A.

The `Partition` function is where the difference between the deterministic and the randomized versions of Quicksort appear. Here, we first consider the deterministic version:

```
//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
}
```

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 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.

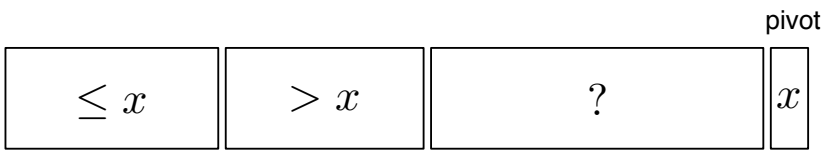

Figure 3: The intermediate state of the `Partition` function: values in the "$\leq x$" region have already been compared to the pivot x, found to be less than or equal in size and moved into this region; values in the "$> x$" region have been compared to x, found to be greater in size and moved into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

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.

2.2.1 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	$x = 6 \;\; p = 4 \;\; r = 5$	$A = [5, \mathbf{6}]$	$[5 \mid \mathbf{6}]$	$[1, 2, 3, 4, 5, 6]$

2.3 Running time

Now that we have proved the correctness of Quicksort, we can analyze how much time and space it uses as it runs. Because Quicksort is a recursive algorithm, we can model it using a *computation tree*, which is a literal representation of the relationship between the various recursive calls to Quicksort. Because Quicksort always makes exactly two recursive calls, every computation tree for Quicksort is a binary tree, where leaf nodes represent base cases and internal nodes represent the recursive calls. Recall our cartoon from above for a maximally unbalanced and a maximally balanced binary tree.

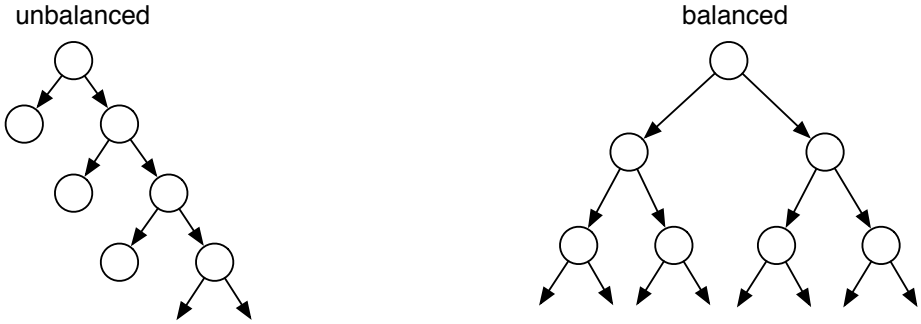

Figure 4: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$.

And, recall that the general form of a simple recurrence relation is

$$T(n) = a\,T(g[n]) + f(n) \;\; , \tag{2}$$

where $T(n)$ is the running time, a is the number of recursive calls made, $g(n)$ is the size of the subproblem passed to each of these calls, and $f(n)$ is the cost of the current call.

2.3.1 Analysis

The loop inside `Partition` examines each element in its subarray and thus `Partition` takes $f(n) = \Theta(n)$ time for a subarray of length n. The total running time of Quicksort thus depends on whether

the partitioning (the recursive step) is balanced or not, and this depends solely on which element in A is used as the pivot. To see why, we'll examine the worst- and best-case performances.

2.3.2 Worst case

The worst performance occurs when the Quicksort recursion tree is maximally unbalanced, which occurs when our partition function always divides A into pieces of size $O(1)$ and $O(n)$ elements.

The recursion cost term (from Eq. (2)) for Quicksort is always $f(n) = \Theta(n)$, because this is the cost we incur for calling `Partition`. For the worst-case division, the function $g[n] = n - b$, for $b = O(1)$. The mathematics for general b are largely the same as for the $b = 1$ case, but the $b - 1$ case is a little easier to understand. The recurrence relation for this worst-case performance of Quicksort is thus

$$\begin{aligned} T(n) &= T(n-1) + T(1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned} \tag{3}$$

The form of $T(n)$ is what we call "tail recursion," which is logically equivalent to a `for` loop. It can easily be solved using the "unrolling" method (sometimes called "substitution"):

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

Thus, the worst-case running time of Quicksort is $\Theta(n^2)$.

(As an at-home exercise, identify the input (an array containing values $x_1, x_2, \ldots x_n$) that produces this worst-case performance? What fraction of permutations of these values lead to this behavior?)

2.3.3 Best case, and the Master Theorem

The best case for Quicksort occurs when the two subarrays are proportionally sized, i.e., when $g[n] = n/b$, with b constant. The mathematics for general b are largely the same as for the $b = 2$

case, in which the pivot is always chosen to be the median of the values in the subarray. In this case, the recurrence relation is

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \ . \end{aligned} \tag{4}$$

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* (CLRS 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. However, if you use it, you need to clearly state the case, and specify the conditions (including a, b, and ϵ).

Before applying it, let's consider what it is 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.[5] 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 $\Theta(f(n))$.[6] [7]

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

[6] 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.

[7] 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.

Applying the master method to Eq. (4) 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. (4).)

3 Quicksort (randomized), and the average case

In our analysis of the worst- and best-case behavior of Quicksort, we argued informally that so long as `Partition` chooses a good pivot element a constant fraction of the time, the average case will be $O(n \log n)$. We will now give a more rigorous treatment of the average case by studying the performance of a randomized variation of Quicksort. To begin, consider these questions:

How often should we expect the *worst case* to occur?
How often should we expect the *best case* to occur?

Because the pivot element is always the last element of each subproblem, i.e., the choice is deterministic, answers to these questions hinge on the precise ordering of the values $x_1, \ldots, x_n$ in the input array.

Recall that for n elements, there are $n!$ possible orderings. For the moment, assume that each of these is equally likely. If we are unlucky and the input array is badly ordered, then we will choose a bad pivot n times and pay $\Theta(n)$ cost each time. Any constant fraction $1/k$ of such bad choices will inflate the depth of the recursion tree by a constant factor k. Being a factor of k deeper does not change the asymptotic performance because k is only a constant, independent of n.

Thus, to fall into the $\Theta(n^2)$ worst case, we need to choose a bad pivot *more* than a constant fraction of the time, e.g., we need to choose poorly a $1 - o(1)$ fraction of the time. The flip side of this argument is that in order to avoid the worst case performance, *we need only choose a good pivot a constant number of times*, regardless of the size of the input. (Do you see why?)

If input orderings are chosen uniformly at random, there is only a small chance of choosing one that will induce such a large number of bad pivots. (Exercise: What is that chance?) By convention, however, we are concerned with worst-case scenarios (as in the "adversarial model" of algorithm analysis), and thus we do not have the luxury to be optimistic about the algorithm's input.

We have already seen that the worst-case performance of Quicksort is fairly inefficient, so this raises a question:

How could we guarantee that the behavior *of Quicksort is close to the average case, even if the* form *of the input is the worst case?*

Decoupling the algorithm's behavior from the input's form, is a powerful trick, and we can achieve it by injecting randomness into the decisions the algorithm makes as it runs. So long as the randomness is independent of the form of the input, this *randomization strategy* will succeed in the decoupling. That is, randomized Quicksort converts an arbitrary (even worst-case) input into a random input, and thus will have an average performance. In the adversarial model of algorithm analysis, this trick defeats the adversary almost surely, because the adversary only gets to control the form of the input, not the choice of random bits used by the algorithm.

3.1 The algorithm

By making use of a pseudo-random number generator to simulate random choices of the pivot element,[8] we can make QuickSort behave as if whatever input it receives is actually an average case. This is our first example of a *randomized algorithm*.[9]

The pseudocode for Randomized Quicksort has the same structure as Quicksort, except that it calls a randomized version of the `Partition` procedure, which we'll call `RPartition` to distinguish it from its deterministic cousin. Here's the main procedure:

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

The `Randomized-Partition` function called above is where this version deviates from the deterministic Quicksort we saw last time:

[8] There is a philosophical debate about whether random numbers can truly be generated that we needn't have. So long as our source of random bits behaves as if its truly random, we can proceed as if it actually is. For the purposes of algorithm analysis, we assume that producing a single random real or integer number is an atomic operation, taking $\Theta(1)$ time.

[9] Randomized algorithms are a big part of modern algorithm theory; see *Randomized Algorithms* by Motwani and Raghavan. In general, a source of random bits—like a pseudo-random number generator—is used to make the behavior of the algorithm more independent of the input sequence, which constrains both the worst- and best-case behaviors. The analysis of randomized algorithms typically requires computing the likelihood of certain sequences of computation using probability theory. Bizarrely, some randomized algorithms can be *derandomized*—removing the source of random bits—even while preserving their overall performance.

```
RPartition(A,p,r) {
   i = random-int(p,r) // NEW: choose uniformly random integer on [p..r]
   swap(A[i],A[r])     // NEW: swap corresponding element with last element
   x = A[r]            // pivot is now a uniformly random element
   i = p-1             // code from deterministic Partition
   for (j=p; j<=r-1;j++) {
      if A[j]<=x {           // same as before
         i++                 //
         swap(A[i],A[j])     //
   }}
   swap(A[i+1],A[r])   //
   return i+1          //
},
```

where `random-int(i,j)` is a function that returns integers from the interval $[i, j]$ such that each integer is equally likely. Thus, we choose a uniformly random element from `A[p..r]` and make *it* the pivot by swapping it with the last element. The rest of the `RPartition` function does exactly the same steps as the deterministic `Partition` function.

3.2 Analysis

Because we are using a source of random bits to randomize the decisions of Quicksort, the running time of Randomized Quicksort is itself a *random variable*. We're interested in the average or *expected* running time—the expected value of this random variable—and to analyze that, we'll need to use a few tools from probability theory. (See Appendix C.2-3 in CLRS, which covers introductory probability theory.)

3.2.1 A few tools from probability theory

A *random variable* is a variable X that takes several values, each with some probability. For example, the outcome of rolling a die, like a pair of 6-sided dice as in some casino games, is often modeled as a random variable. The *expected value* of a random variable is defined as

$$E(X) = \sum_x x \Pr(X = x) \qquad \text{given that } \sum_x \Pr(X = x) = 1 \text{ and } x \text{ is discrete} \tag{5}$$

$$E(X) = \int_x x \Pr(x) \mathrm{d}x \qquad \text{given that } \int_x \Pr(x) \mathrm{d}x = 1 \text{ and } x \text{ is continuous.} \tag{6}$$

The right-hand sides of these statements verify that the function $\Pr(x)$ in each case is a *probability distribution*. For example, the expected value of a single roll of one 6-sided die is

$$\begin{aligned} E(X) &= 1\left(\frac{1}{6}\right) + 2\left(\frac{1}{6}\right) + 3\left(\frac{1}{6}\right) + 4\left(\frac{1}{6}\right) + 5\left(\frac{1}{6}\right) + 6\left(\frac{1}{6}\right) \\ &= \frac{1}{6}\sum_{i=1}^{6} i = \frac{1}{6}\cdot\frac{6(6+1)}{2} = \frac{21}{6} = 3.5 \ . \end{aligned}$$

In this example, outcomes are *independent*, which means that the outcome of one roll does not change the likelihood of outcomes on any subsequent roll. Mathematically, we say that if $X = x$ and $Y = y$, x and y are independent if and only if ("iff"):

$$\Pr(X = x, Y = y) = \Pr(X = x)\Pr(Y = y) \ . \tag{7}$$

Continuing the dice example, the probability of getting "snake eyes", that is, both dice show a 1, is $\Pr(X = 1, Y = 1) = \Pr(X = 1) \cdot \Pr(Y = 1) = 1/6 \cdot 1/6 = 1/36$. That is, the probability of getting $X = 1$ *and* $Y = 1$ is the product of the probabilities of the individual events.

Similarly, the probability of getting $X = 1$ *or* $Y = 1$ is the sum of the individual probabilities. For example, given two dice, the probability of getting at least one of the dice to show a 1 is $\Pr(X = 1 \text{ or } Y = 1) = \Pr(X = 1) + \Pr(Y = 1) = 1/6 + 1/6 = 1/3$.[10]

An *indicator random variable* is a special kind of random variable, which use can use to count the number of times some event A occurs:

$$I(A) = \begin{cases} 1 & \text{if the event } A \text{ occurs} \\ 0 & \text{otherwise} \ . \end{cases}$$

For instance, we could use such a structure to count the number of times a 6-sided die comes up 1.

If X and Y are two random variables, then the property of *linearity of expectations* states that

$$E(X + Y) = E(X) + E(Y) \ , \tag{8}$$

which holds even if X and Y are not independent. More generally, if $\{X_i\}$ is a set of random variables, then by linearity of expectations, the following is true:

$$E\left(\sum_i X_i\right) = \sum_i E(X_i) \ . \tag{9}$$

[10] For a fun excursion into combinatorics and probability, try computing the probability of getting a *yahtzee*, i.e., all dice showing the same value, for 5 dice where you may re-roll each die at most 2 times after your initial roll. Hint: the probability is several thousand times larger than $6 \times (1/6)^5 \approx 0.00077$.

3.2.2 Analyzing Randomized Quicksort

With these tools, we can now analyze the running time of Randomized Quicksort.

Let the input array be denoted $A = [z_1, z_2, \ldots, z_n]$ and let A be already sorted, that is, z_i is the ith smallest element of A. Further, let the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$ denote those elements between z_i and z_j, inclusive. Note that A being sorted has no impact the running time of Randomized Quicksort.

When does Randomized Quicksort compare z_i and z_j? This comparison will occur either one or zero times, and it occurs only when (i) a subproblem of Randomized Quicksort contains Z_{ij} *and* (ii) either z_i or z_j is chosen as the pivot element. Otherwise, z_i and z_j are never compared. (Do you see why?) Let X denote the running time of Randomized Quicksort; X is then exactly equal to the number of times z_i and z_j are compared, for all i, j, i.e., X counts the number of comparison operations.[11] That is,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ ,$$

where we define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$.

By linearity of expectations, we can simplify this expression:

$$\begin{aligned} E(X) &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) \ . \end{aligned} \tag{10}$$

This is more helpful, but Eq. (10) is not something we can express yet in big-O notation. To do that, we need to further refine what $\Pr(z_i \text{ is compared to } z_j)$ means. Recall that

$$\Pr(z_i \text{ is compared to } z_j) = \Pr(\text{either } z_i \text{ or } z_j \text{ are the first pivots chosen in } Z_{ij}) \ .$$

To see that this is true, recall two facts:

[11] The total number of atomic operations is proportional to the number of comparisons, and thus, for asymptotic analysis, it is sufficient to count only the comparisons.

1. if no element in Z_{ij} has yet been chosen, no two elements in Z_{ij} have yet been compared, and thus all of Z_{ij} are in the same subproblem of Randomized Quicksort; and
2. if some element in Z_{ij} other than z_i or z_j is chosen first, then z_i and z_j will be split into separate lists, and thus will never be compared.

To complete our analysis, we need to identify $\Pr(z_i \text{ is compared to } z_j)$ in terms of i and j, the summation indices in Eq. (10). Recall that the choice of z_i is independent of z_j and that we choose the pivot uniformly from the set Z_{ij}, which has $j - i + 1$ elements. Thus, we can write

$$\begin{aligned}
&\Pr(z_i \text{ is compared to } z_j) \\
&\quad = \Pr(\text{either } z_i \text{ or } z_j \text{ is first pivot chosen in } Z_{ij}) \\
&\quad = \Pr(z_i \text{ is first pivot chosen in } Z_{ij}) + \Pr(z_j \text{ is first pivot chosen in } Z_{ij}) \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&\quad = \frac{2}{j-i+1} \ . \qquad (11)
\end{aligned}$$

Substituting this result into Eq. (10), we can now complete our analysis

$$\begin{aligned}
E(X) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) \ . \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} && \text{change of variables } k = j - i \qquad (12) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} && \text{lower bound} \qquad (13) \\
&= \sum_{i=1}^{n-1} O(\log n) && \text{harmonic series bound} \qquad (14) \\
&= O(n \log n) \ . \qquad (15)
\end{aligned}$$

Thus, the expected running time of Randomized Quicksort is $O(n \log n)$. This result is also a proof of the expected running time for (deterministic) Quicksort, when inputs are equally likely.

3.3 An alternative analysis, and a trick

There are several other ways to analyze the performance of Randomized Quicksort. Here is one that doesn't use any of the probabilistic tools we used above, but instead uses some tricks from

recurrence relations. Let's consider the specific structure of the splits that `Partition` induces. Recall that generally speaking, the running time is given by the recurrence

$$T(n) = T(k-1) + T(n-k) + \Theta(n) \ , \tag{16}$$

where the value $k-1$ is the number of elements on the left side of the recursion tree, i.e., k tells us how good (balanced) the split is. If $k = 1$ then we have the worst case and if $k = n/2$ we have the best-case. More generally, if $k = O(1)$ the running time is $\Theta(n^2)$ and if $k = O(n)$ the running time is $O(n \log n)$. In Randomized Quicksort, all values of k are equally likely and thus the average running time is given by averaging Eq. (16) over all k:

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{k=1}^{n} \left(T(k-1) + T(n-k) + \Theta(n)\right) \\ &= c_1 n + \frac{1}{n}\sum_{k=1}^{n} T(k-1) + \frac{1}{n}\sum_{k=1}^{n} T(n-k) \ , \end{aligned}$$

where we have made the asymptotic substitution $\Theta(n) = c_1 n$, for some constant c_1. We can simplify this by substituting $i = k-1$ in the first summation and $i = n-k$ in the second, and then multiplying through by n:

$$\begin{aligned} T(n) &= c_1 n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \\ n\,T(n) &= c_2 n^2 + 2\sum_{i=0}^{n-1} T(i) \ . \end{aligned} \tag{17}$$

(Do you see why the summation bounds change the way they do?) Now we'll use a technique you can use to solve some recurrence relations: subtract a recurrence for a problem of size $n-1$ from the recurrence for a problem of size n. Using Eq. (17), this yields:

$$\begin{aligned} n\,T(n) - (n-1)T(n-1) &= \left(c_2 n^2 + 2\sum_{i=0}^{n-1} T(i)\right) - \left(c_2 (n-1)^2 + 2\sum_{i=0}^{n-2} T(i)\right) \\ n\,T(n) - (n-1)T(n-1) &= 2T(n-1) + c_2(2n-1) \\ n\,T(n) &= (n+1)T(n-1) + c_2(2n-1) \ , \qquad (18) \\ \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{c_2(2n-1)}{n(n+1)} \ . \qquad (19) \end{aligned}$$

Where we divided both sides of Eq. (18) by $n(n+1)$ to produce Eq. (19).

Now, one last change of variables: let $T(n)/(n+1) = D(n)$, and thus $T(n-1)/n = D(n-1)$. This yields:

$$\begin{aligned} D(n) &= D(n-1) + O(1/n) \\ &= O(\log n) \ , \end{aligned}$$

where the last line follows from a bound on the harmonic series. Thus, reversing the substitution for $D(n)$, we see that $T(n) = O(n \log n)$.

3.4 Alternative Quicksorts

There are other ways to change Quicksort to avoid the worst-case behavior, all of which focus on choosing good pivot elements; some of these are discussed in the textbook.

The most popular of these alternatives is to use a `Median-of-k` algorithm to choose the pivot element. In this version, the `Partition` algorithm selects the pivot as the *median* of k elements, where typically $k = O(1)$ ($k = 3$ is a common choice). The larger k, the closer the identified element is to the true median of a particular subproblem. The closer the pivot is to the true median, the more balanced a division is made. This strategy further reduces the number of input sequences that yield a worst-case performance. (What happens if we set $k = n$ in this algorithm?)

4 On your own

1. Chapters 4 and 7 in CLRS