CS584/684 Algorithm Analysis and Design – Spring 2017
Week 1: Introduction; Selection

This course provides an "advanced in-depth study" of algorithm analysis and design. This implies that it follows on from a more basic and/or shallow introductory course, such as PSU CS350, in which you encountered the "standard" deterministic, sequential algorithms for sorting, searching, graph traversal, etc., and learned basic techniques for studying their asymptotic complexity. (This is roughly the material in Parts I–III of the CLRS textbook.)

So, we won't be covering this material again; instead we will leap directly into more exciting topics. But I understand that you might be feeling a bit rusty about the basics, so I'll try to incorporate the opportunity to review relevant background (especially as part of the homework) as we go along. Today's (non-counting) quiz will help you get a feeling for how much review you may need to do.

There are several different themes running through the topics in this course.

Algorithmic Strategies

- Brute force. This is somewhat ill-defined, but it basically means a "dumb" approach that doesn't take advantage of any special structure of the problem. In particular, we can often solve a problem by enumerating all elements in the space of potential solutions and testing each one. If that space is large, this approach may be hopelessly inefficient, but it can still be a useful first step.
- Divide and Conquer. Solve a problem by first decomposing it into smaller sub-problems, then recursively solve the sub-problems, and finally compose the sub-solutions into an overall solution. An amazingly effective approach to many problems, if the decomposition/composition costs can be kept down and the size of the sub-problems is well-balanced.
- Randomization. Explicitly randomize the behavior of an algorithm in order to compensate for potentially unfriendly inputs. This can sometimes make big improvements in the expected running time of the algorithm, and it is often quite simple and practical to do.
- Reduction. In practice, most problems can be solved by using one or more existing algorithms. In this situation, we solve a problem P by *reducing* it to a problem R that has a known solution. Obviously, you can only do this if you know about R and its solution, so it is important to gain a wide knowledge of the useful algorithms that are out in the world. Reductions can sometimes make surprising bridges from one application domain to another, apparently unrelated, domain. Reduction is also an important concept for showing the *hardness*, or even impossibility, of certain problems; if we can reduce P to R, and P is already known to be hard, then R must be at least as hard.
- Dynamic programming. Memoize solutions to sub-problems that occur repeatedly (a way of trading time for space).
- Greediness. Take advantage of problems where a locally good solution is also a globally good one.
- Etc., etc. This is by no means an exhaustive list. Algorithm design is a playground for creativity!

Models of Computation

To implement and analyze an algorithm, we need to pin down what computational resources are available. This is particularly important when trying to establish *lower bounds* on the cost of solving a problem.

- Random Access Machine (RAM). The standard model for sequential computation. We measure time and space by counting "instructions" and "memory cells" on a pseudo-machine resembling a real Von Neumann processor. We normally assume uniform cost to access any memory location.
- Parallel Random Access Machine (PRAM). A parallel version of the RAM model suitable for synchronous algorithms.
- Work-Span model. A model for describing asynchronous multithreaded algorithms characterized by *work* (number of computation steps) and *span* (longest dependency chain).

Correctness

One of our biggest concerns should be that our algorithms actually solve the problems they claim to. Algorithms courses are often rather informal about proving correctness, especially because the algorithms may be given in imprecise pseudo-code. But it is important to be able to get formal when we need to assure ourselves that complicated code behaves correctly.

Our approach to proving correctness is based on the structure of the algorithm.

- For each procedure in the algorithm, especially recursive ones, we can specify logical *pre-conditions* and *post-conditions* for each procedure and prove that for any execution of the procedure, if we require that the pre-conditions are true at procedure entry, then the code ensures that the post-conditions at procedure exit. Moreover, we prove that at each procedure call, the calling code establishes the pre-conditions; this allows us to assume that the post-conditions hold after return to the caller.
- For iterative algorithms, we use the method of *loop invariants*. An invariant is a logical specification that is true at loop entry, maintained by each loop body execution, and (hence) true at loop exit. We design the invariant so that it implies useful information after loop exit, which can be used to prove the post-condition of the surrounding procedure.
- In addition to proving that post-conditions are met, we typically also want to make sure that the code actually terminates. This is usually done by showing that on each recursive call and each loop iteration, some (non-negative) measure on the program state gets smaller.

Resource Analysis

How much time and space, and (for parallel programs) how many processors does an algorithm require on an input of a given size? This kind of analysis is at the heart of the course. We'll rely on a number of techniques:

- Recurrences. Computing the cost of recursive programs naturally leads to recurrence relations. We will use a variety of mechanisms for finding and verifying closed-form solutions to recurrences, including proof by induction (the "substitution method"), drawing recurrence trees, and applying generic rules (such as the "master theorem").
- Probabilistic analysis. We can use techniques from finite probability theory to bound the "average case" cost of deterministic algorithms or the "expected" cost of randomized algorithms, which may be much more important than the worst-case cost.

- Amortized analysis. Suppose we execute a sequence of operations on a data structure. Even if the worst-case cost of a single operation is high, the average cost per operation of the whole sequence might be much lower. Amortized analysis can be used to obtain good bounds for the overall sequence.
- Experimentation. Sometimes the math looks too hard (either because we don't know enough or because it really is). In such cases, it can be helpful to simply program up the problem—either an algorithm or an analysis calculation—to explore what happens at (relatively) small sizes and look for patterns.

Asymptotics

We are especially interested in *asymptotic analysis*, i.e. the behavior of programs as problem size grows towards infinity. Why? Because asymptotic behavior is the usually the best indicator of which algorithm to pick, except for very small inputs. (Of course, "very small" here depends on the size of the constants hidden by the asymptotic notation.)

Notation definitions (for CLRS, at least):

$f(n) = O(g(n))$ means there exist $n_0, c > 0$ such that $\forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$.

$f(n) = \Omega(g(n))$ means there exist $n_0, c > 0$ such that $\forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$.

$f(n) = \Theta(g(n))$ means $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

One good way to get a grip on the importance of rates of growth is to consider the following. Suppose computer speed doubles every 18 months (as it used to in the good old days of Moore's law), and that every 18 months we go out and buy a new processor (as a certain local company dearly hopes we will). If we could solve a problem of size 10^6 in one hour on our old machine, how big a problem can we solve in one hour on our new machine?

Algorithm Speed	Old Machine	New Machine
$\lg n$	1000000	$\approx$ 1100000000000
$\sqrt{n}$	1000000	4000000
n	1000000	2000000
$n \lg n$	1000000	$\approx$ 1920000
n^2	1000000	1414213
n^3	1000000	1259921
2^n	1000000	1000001
$n!$	1000000	1000000

Problem: Median Finding

"Given an array A of size n containing integers in arbitrary order, find the *median element*, i.e. the value x such that half the elements of A are greater than x and half are less than x." For simplicity, we will assume from now on that the elements of A are distinct (no repetitions) and that n is odd (otherwise there are really two medians). So the median is the dth smallest element, where $d = \lfloor n/2 \rfloor + 1$. (By the way, problems like this are often informally posed as "given a *set*" or "*sequence*" of elements, but for the algorithms we're looking at here, it is actually crucial that we be given an array. We will look at the general issue of data representation more later.)

As a simple example, the median of the array

7	3	-2	6	-1	4	9	-5	5

is 4.

How shall we solve this problem?

Brute force. Given a candidate median x, it is straightforward to check whether it is the median: we compare x with every other element of A and calculate c = the number of elements in A that are are less than x; x is the median iff $c = \lfloor n/2 \rfloor$. So we can find the median by considering every element of A in turn as a candidate x and testing it until we find the winner. But this takes $\sum_1^n (n-1) = \Theta(n^2)$ worst-case time: an unsatisfactory result.

Reduction. Suppose A were sorted in increasing order. Then we can read off the median directly: it is just $A[d]$. But we already know how to sort arrays! We can therefore reduce median finding to sorting, and inherit the efficiency characteristics of the sort. For example, we can use Mergesort to do the job in worst-case $O(n \lg n)$ time (but $O(n)$ extra space), or use Quicksort to get "average-case" $O(n \lg n)$ time and worst-case $O(n^2)$ time (but only $O(1)$ extra space), or use Heapsort to get [what?], or ... At any rate, we can solve the median problem in $O(n \lg n)$ worst-case time. And this is the best we can do by reducing to comparison-based sorting, because we know a $\Omega(n \lg n)$ lower-bound for that. (Remember decision trees?)

Divide-and-Conquer. But can we do better? Sorting obviously discovers enough information to solve the median problem, but perhaps it discovers more than enough. Can we find a more direct, and perhaps more efficient, divide-and-conquer algorithm?

As inspiration for a divide-and-conquer algorithm, recall how Quicksort on arrays works: it chooses an arbitrary pivot, *partitions* the array into subarrays whose elements are all less than, resp. greater than the pivot, and then sorts both subarrays recursively. The partition function looks something like this (CLRS p. 171; this version of partition is due to Nico Lomuto):

```
PARTITION($A, p, r$)
    // Requires on call: $1 \le p \le r \le A.length$.
    // Ensures at return:
    //   $A[ret] = A'[r]$, where $ret$ is the return value, and $A'$ is the array as it was at entry to the function.
    //   $\forall k \in [p\,..\,ret-1] : A[k] \le A[ret]$ and $\forall k \in [ret+1\,..\,r] : A[k] \ge A[ret]$.
    //   $\mathrm{PermOn}(A, A', p, r)$, meaning that $A[p\,..\,r]$ is a permutation of $A'[p..r]$ and $\forall k \notin [p\,..\,r], A[k] = A'[k]$.
1   $x = A[r]$
2   $i = p - 1$
3   for $j = p$ to $r - 1$
    //   $\forall k \in [p\,..\,i], a[k] \le x$ and $\forall k \in [i+1\,..\,j-1], a[k] > x$
    //        and $j \in [p\,..\,r]$ and $i \in [p-1\,..\,j-1]$ and $\mathrm{PermOn}(A, A', p, j-1)$
4       if $A[j] \le x$
5           $i = i + 1$
6           exchange $A[i]$ with $A[j]$
7   exchange $A[i+1]$ with $A[r]$
8   return $i + 1$
```

(By the way, throughout the course we will follow the conventions of CLRS by starting array indices from

1 rather than 0 in our pseudo-code—but not, of course, in practical programming assignments in C or Java or Python or other languages that use 0-based indexing.)

Here, the last element of the inital array, $A[r]$, is used as the pivot value. As an example, if we run the algorithm on this array:

i	p, j								r
	7	3	-2	6	-1	4	9	-5	5

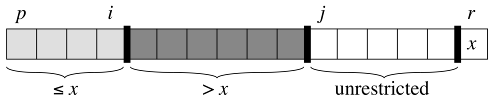
we end up with this one:

p				i				j, r
3	-2	-1	4	-5	5	6	9	7

and $i + 1$ is the position of the pivot value.

This certainly seems to work, but it may not be at all obvious why! To show correctness of the algorithm (with respect to the pre-conditions and post-conditions recorded in the comment at the top of the code), and indeed, even to understand it informally, we need to study the *invariant* of the **for** loop at line 3, which is a relationship between the contents of the array and the four index variables p, i, j, r. (We will often write $[a \,..\, b]$ for the set containing integer values k such that $a \leq k \leq b$; this is a slight extension from CLRS, which uses this style only to denote subarrays. Note the set $[a \,..\, b]$ is empty if $b < a$, in which case anything we say about its elements is vacuously true.)

This invariant is most easily understood using a picture (CLRS Figure 7.2):



Essentially, j maintains the boundary between the set of elements we have already partitioned with respect to x and set of those we have not yet considered and i maintains the boundary between elements known to be $\leq x$ and those known to be $> x$. The invariant is certainly true on initial entry to the loop, because at that point the two sets $[p \,..\, i]$ and $[i + 1 \,..\, j - 1]$ are empty and array A has not been changed at all. The key to understanding the algorithm is seeing how it *maintains* the invariant at each loop iteration. We need to move $A[j]$ from the unprocessed set to the correct part of the processed set. If $A[j] > x$, this can be done trivially just by advancing j while leaving i alone. If $A[j] \leq x$ we need to advance i and place the value $A[j]$ somewhere within the $A[p \,..\, i]$ subarray before advancing j. The exchange accomplishes this neatly: we move the old $A[i]$ out of the way, put $A[j]$ in its place, and store the old $A[i]$ into the now free $A[j]$ slot. (The net effect is to change the old $A[i]$ value from being the leftmost member in its set to being the rightmost one.) The remaining clauses of the invariant enforce that i and j remain within bounds at each array read and write operation, and that the new contents of $A[p \,..\, j - 1]$ is a permutation of the original contents and the remainder of the array is not changed. Finally, the loop terminates after we have processed all elements except $A[r]$. At that point, we do one last exchange to put $A[r]$ into its proper place just to the right of all values known to be smaller. Now the post-condition of the function is true.

In the given example, the partitioning element ends up roughly in the middle of the result array. But of course, this won't always be the case. For example, if the initial value of $A[r]$ happens to be the minimum or maximum value in the array (e.g. if the inital array is sorted in increasing or decreasing order), we will end up with a maximally unbalanced partitioning. It is well worth working through the details for such cases. (When doing this, note that it is perfectly possible to have $i = j$ at line 6, causing a vacuous "exchange" between an element and itself.)

It is easy to verify that this procedure has aymptotic execution time $\Theta(n)$ where $n = r - p + 1$. Note in particular that although the number of swaps performed by the algorithm depends on the contents of the array, this does not affect the asymptotic time, since each element has to be inspected regardless and swapping adds only a constant additional cost.

Revised Problem: Selection We will use PARTITION as a subroutine in an algorithm for the median finding problem. But first, we need to *generalize* the problem slightly. This is often a helpful step; although superficially it seems to make the problem harder, it often exposes structure that can be used to find a recursive solution (and an inductive proof of correctness). Here the generalization we want is go from the median finding problem to the arbitrary *selection* problem:

"Given an array A of size n containing integers in arbitrary order and a value i with $1 \leq i \leq n$, find the ith smallest value in A."

Again, we will assume that the elements of A are distinct. Obviously, we can reduce median finding to selection by specifying $i = d$. Using PARTITION as a subroutine, we can now give a direct divide-and-conquer algorithm, which we call QUICK-SELECT. Here is the code:

```
QUICK-SELECT($A, p, r, i$)
    // Requires on call: $1 \leq p \leq r \leq A.length$ and $1 \leq i \leq (r - p + 1)$ and elements of $A$ distinct.
    // Ensures on return: $|\{x \in A[p\,.\,.\,q] \mid x < ret\}| = i - 1$ and $\mathrm{PermOn(A, A', p, q)}$.
1   if $p == r$
2       return $A[p]$
3   $q = $ PARTITION($A, p, r$) // $A[r]$ is the pivot value
4   $k = q - p + 1$ // $k$ is the size of the lower partition
5   if $i == k$
6       return $A[q]$
7   elseif $i < k$
8       return QUICK-SELECT($A, p, q - 1, i$)
9   else
10      return QUICK-SELECT($A, q + 1, r, i - k$)
```

As in Quicksort, the procedure begins by invoking PARTITION to partition the sub-array $A[p\,.\,.\,r]$ into two parts around a pivot value $A[r]$, returning the pivot position q. (We will discuss choice of pivot in detail later; for now it is simply whatever happens to already be the last value in the sub-array.) After partitioning, we know that all elements in the lower partition ($A[p\,.\,.\,q-1]$) are less than x and and all those in the upper partition ($A[q+1\,.\,.\,r]$) are greater than x. The key idea is that just by looking at the *size* k of the lower partition, the code can determine which partition contains the ith smallest element of $A[p\,.\,.\,r]$, and invoke itself recursively on that partition. If the recursive invocation is on the upper partition, the index argument i must be adjusted to account for the size of the lower partition. The recursion terminates when the pivot is itself the ith element, or when the sub-array being considered has size 1. (Termination is guaranteed because the sub-array shrinks by at least one element at each recursive call.) To get the whole thing started to find the ith element of array A, we simply invoke QUICK-SELECT($A, 1, A.length, i$).

Running Time

Observe that whereas Quicksort calls itself recursively on *both* partitions, QUICK-SELECT does so on only *one* partition. We might hope, therefore, that it will run faster than Quicksort, in the sense of having worst-case time $o(n^2)$. (Remember "little-oh" notation?). Alas, this is not the case.

Let $T(p, r)$ be the worst-case running time for QUICK-SELECT on the sub-array $A[p\,..\,r]$. The two components of this are the call to PARTITION, which takes linear time, and the recursive call: since we don't know which partition will be used for the recursive call, we assume the worst case, i.e. the larger partition. This gives:

$$T(p, r) = \Theta(r - p + 1) + T(\max_{q \in [p \ldots r]}(q - p, r - q)).$$

Now we can see that if we are very unlucky with our input (e.g. if the initial array is sorted), we will repeatedly end up with a q value that is equal to p or r, so that the recursive call will be on a sub-array just one element smaller than before. Rewriting in terms of the sub-array size n and picking a constant a to approximate the linear term, we get

$$T(n) = an + T(n - 1)$$

Expanding this out (and taking $T(0) = 0$) gives an arithmetic series:

$$T(n) = an + a(n-1) + a(n-2) + \ldots + 0 = a\left(\sum_{i=0}^{n} i\right) = a\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

This seems quite unpromising, since we already know how to use reduction to sorting to get $O(n \lg n)$ worst-case time (assuming we use a $O(n \lg n)$ worst-case sorting algorithm, such as Mergesort).

However, all is by no means lost. It turns out that we just need to pay closer attention to the selection of the pivot value. The key idea is that we need to choose pivots so as to guarantee that each recursive call occurs on a significantly smaller sub-array than its parent. What does "significant" mean? Well, suppose we know that the size of the sub-array shrinks by at least a *constant* factor on each recursive call, i.e.

$$\max(q - p, r - q) \leq c(r - p + 1)$$

for some constant c with $0 < c < 1$. Now we get

$$T(n) \leq an + T(cn)$$

Expanding this out for L levels of recursive calls gives

$$T(n) \leq an + acn + ac^2n + \ldots = an\left(\sum_{i=0}^{L} c^i\right) \leq an\left(\sum_{i=0}^{\infty} c^i\right) = an\left(\frac{1}{1-c}\right) = O(n)$$

Note that if we just need an asymptotic upper bound, we don't need to bother to figure out what L is (although that shouldn't be too tough for you); it is often easier to simply pass to the limit as we did here. Voila! By the magic of the geometric series, we get selection in linear time!

But. . .how *can* we obtain a constant factor reduction? We will examine two approaches:

• Use a *randomized* algorithm that simply picks a random pivot position at each recursive call. This leads to a linear *expected time* algorithm. This algorithm is quite practical and usually behaves well (regardless of the distribution of the actual input).

• Find a *deterministic* way to compute a suitable pivot. It turns out that there is such a way, which is quite non-obvious! It is complicated and not very practical (the constants hidden by the O-notation are large), but is worth seeing.

We'll start with . . .

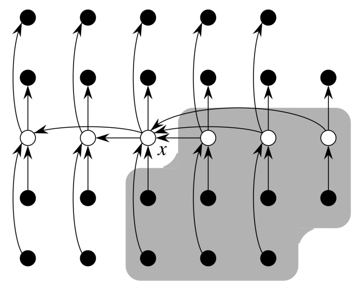
Median-of-medians

This deterministic algorithm was published in 1972 by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan (four out of five are Turing award winners!).

MOM-SELECT is a variant of QUICK-SELECT in which the computation of the pivot value at line 3 is preceded by the following steps:

1. Divide the n elements of the array into $\lceil n/5 \rceil$ groups, each with 5 elements (except perhaps the last one). (Note: The choice of 5 is not arbitrary, but it is not the only possible value to use, as you'll explore in the homework.)
2. Find the median of each group of 5 by any method (e.g. reduction to sorting, or even brute force).
3. Invoke MOM-SELECT recursively on the $\lceil n/5 \rceil$ medians to obtain pivot value x (the "median of medians").
4. Determine the index i of x in the array and exchange $A[r]$ with $A[i]$ (so x will be in the pivot position).

We claim that this method of pivot selection guarantees that the larger of the resulting partitions is a constant factor smaller than we started with. The proof is mainly by a picture (CLRS Figure 9.1):

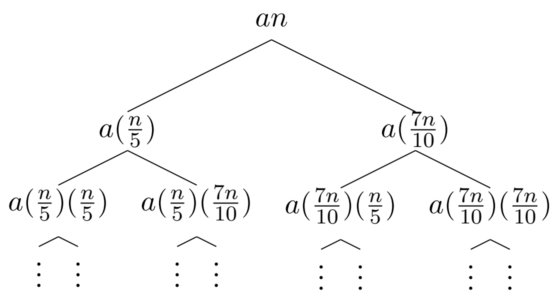


The diagram shows possible relationships between values just before a recursive call. Each circle represents a value in the array of n elements. The vertical columns are the $\lceil n/5 \rceil$ groups of 5. The white circles are the medians of the groups, and x is the median of medians. There is an arrow from one circle y to circle z to another when we know for sure that $y > z$. (On the other hand, the absence of an arrow tells us nothing.) The shaded background delimits the set of values that we know to be greater than x. The key point is that this set represents at least a constant fraction of the entire array. This can be shown by a little counting: roughly speaking, at least half of the columns contribute at least 3 elements to this set, so it has at least $\frac{3n}{10}$ elements. (The book gives a more pedantic bound.) Dually, at least $\frac{3n}{10}$ elements are known to be less than x. Hence, whichever recursive call we make, it will be on a sub-array of size *at most* $\frac{7n}{10}$. In other words, we seem to have established our desired constant $c = \frac{7}{10}$.

To ensure that MOM-SELECT runs in linear asymptotic time, there is actually a bit more work to do, though. We must examine our new three-step process for picking the pivot contributes to see what it adds to the running time. The details of steps 1 and 4 are a little sketchy, but each can surely be done in $O(n)$ time. Crucially, step 2 can be also be done in $O(n)$ time: because each of the $\lceil n/5 \rceil$ groups has at most 5 elements, we can find its median in *constant time* regardless of the particular method used. But we have to track the cost of the recursive call in step 3, which will be on roughly $n/5$ elements. So the recurrence we need to solve is actually:

$$T(n) \leq an + T(n/5) + T(7n/10)$$

To confirm that this still leads to a linear overall time, we can use a *recursion tree*, which has a node corresponding to each recursive call. Each node is labeled with the local (non-recursive) cost of the call; the total cost is obtained by summing the cost at each level and then summing over all levels. The shape of the tree for this recurrence is:


an
$a(\frac{n}{5})$
$a(\frac{7n}{10})$
$a(\frac{n}{5})(\frac{n}{5})$
$a(\frac{n}{5})(\frac{7n}{10})$
$a(\frac{7n}{10})(\frac{n}{5})$
$a(\frac{7n}{10})(\frac{7n}{10})$

The level sums are $an, a(9/10)n, a(81/100)n, \ldots, a(9/10)^L n$. As before, the simplest thing is to let the number of levels extend to infinity, yielding

$$T(n) \leq an\left(\sum_{i=0}^{\infty}\left(\frac{9}{10}\right)^i\right) = an\left(\frac{1}{1-(9/10)}\right) = 10an = O(n)$$

Conclusion: Moms rule!

The randomized approach

In this approach, we use a random number generator to pick each pivot uniformly at random from the sub-array. Specifically, we prefix line 3 of QUICK-SELECT by

```
Exchange A[RANDOM(p,r)] with A[r].
```

where RANDOM is a $\Theta(1)$-time utility function that returns a (pseudo-)random integer in the range $[p \mathinner{.\,.} r]$. By picking the pivot at random, we expect that the partition sizes will be fairly balanced on average, and hence that the revised algorithm runs in expected linear time, *regardless* of the order of values in the input array. We'll proceed to show this by probabilistic analysis.

Note that an alternative approach would be to use a similar probabilistic analysis to show that the *unrandomized* QUICK-SELECT algorithm runs in linear time in the "average" case *assuming* that input array order is random, i.e. that every ordering of the values is equally likely. But this would be a weaker result because it relies on a stronger assumption: we often don't know anything about the distribution of orderings for our input arrays, much less that it is suitably random. Indeed, in practice, input arrays are often quite non-randomly ordered (e.g., they might be in, or nearly in, increasing or decreasing order).

Let's again write n for $r - p + 1$, the size of the sub-array being partitioned at any given recursive step. First of all, note that since any pivot argument to PARTITION is equally likely, with probability $1/n$, the resulting pivot position q after partitioning is also evenly distributed, i.e., for each $i \in [p \mathinner{.\,.} r]$, $\Pr\{q = i\} = 1/n$.

In a probabilistic setting, we treat $T(n)$ as a *random variable* representing the running time on a problem of size n, and our goal is find the *expected value* $E[T(n)]$. (Aside/Reminder: a random variable X is just a real-valued function defined over the some probability sample space. The expected value of X is the average value of X over all values it takes on, defined as $E[X] = \sum_x x \cdot \Pr\{X = x\}$. An important feature of expected values is that $E[X + Y] = E[X] + E[Y]$, whether or not X and Y are independent. On the other hand $E[XY] = E[X]E[Y]$ does *not* hold in general, although it does hold if X and Y are independent. As an important special case, $E[X^2] = (E[x])^2$ does *not* hold in general.)

There are now several ways we might proceed. We will show two approaches that work, and one that doesn't.

Indicator Random Variables. This is the approach used by CLRS. The basic idea is to introduce a set of random variables to describe the possible sizes k for the left partition after PARTITION returns. Specifically, for each $k \in [1 \,..\, n]$, let

$$X_k = \begin{cases} 1 & \text{if the subarray } A[p \,..\, q] \text{ has exactly } k \text{ elements} \\ 0 & \text{otherwise} \end{cases}$$

(Aside: The X_k are called *indicator* random variables: they encode a boolean condition by taking on just one of two values: 1 if the condition is true and 0 if it is false.)

From our previous discussion, we can see that $E[X_k] = 1/n$ for all k. Moreover, since X_k can only be non-zero for one choice of k, we have $\sum_{k=1}^{n} X_k = 1$. When $X_k = 1$, the two arrays on which we might recurse have sizes $k-1$ and $n-k$, and we are need to track the *larger* of these. Putting these things together, we obtain

$$\begin{aligned} T(n) &\leq \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + an) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + an \end{aligned}$$

Taking expected values, we get

$$\begin{aligned} E[T(n)] &\leq E\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + an\right] \\ &= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + an && \text{(linearity of expectation)} \\ &= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + an && (*) \\ &= \frac{1}{n} \cdot \sum_{k=1}^{n} E[T(\max(k-1, n-k))] + an && \text{(factoring out known expectation)} \\ &= \frac{2}{n} \cdot \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + an && \text{(by some algebra involving max)} \end{aligned}$$

where the line marked (*) is justified because X_k is independent of $E[T(\max(k-1, n-k))]$, i.e. because the choice made by one call to PARTITION is independent of the choices made by later calls.

Unfortunately, it is not obvious what the solution of this recurrence is and the usual techniques for producing quick rough estimates of solutions do not readily apply. However, if we simply make the (optimistic?) guess that the solution is $O(n)$, it is straightforward (though tedious) to verify this by induction (see CLRS for details).

Estimating the rate of shrinkage. Rather than use one random variable for each possible value of q, it might seem like a simpler approach would be to use a *single* random variable M that describes the size of the bigger result partition, i.e. the value of $\max(q-p, r-q)$. A seemingly natural place to start is to compute the *expected value* $E[M]$.

It's easy to guess the answer by pretending the problem is continuous. If you divide a candy bar randomly into two pieces, the average size of the *bigger* piece will be 75% of the original. (Why? Think about it this way: half the time the left piece will be bigger, and its size will be distributed randomly between 50% and 100% of the original, so averaging 75%. The other half of the time the right piece will be bigger, again with an average size of 75%. Averaging the two cases, we again get 75%.) So that's what we expect here.

Here's a more formal argument using discrete probability. We have

$$\begin{aligned}
E[M] &= \sum_{i=p}^{r} \Pr\{q = i\} \cdot \max(i-p, r-i) \\
&= \frac{1}{n} \sum_{i=p}^{r} \max(i-p, r-i) && \text{(factoring out known probability)} \\
&= \frac{1}{n} \sum_{j=0}^{n-1} \max(j, n-1-j) && \text{(by change of variables)} \\
&\leq \frac{1}{n} \cdot 2 \cdot \sum_{j=n/2}^{n-1} j && \text{(by some algebra involving } \max\text{)} \\
&\leq \frac{3}{4} \cdot n && \text{(by the formula for arithmetic sums)}
\end{aligned}$$

This seems like good news, since it means the expected value of the ratio of recursive-call array size to original array size is $E[C] = E[M]/n = \frac{3}{4}$, i.e. a constant smaller than 1, as we need for our overall time bound.

But there is a difficulty: just because the expected value of C is a suitably small constant doesn't mean that *every* recursive invocation actually shrinks the problem size by that constant—some calls might shrink it by less, others by more. In fact, on average, the problem shrinks by $\frac{3}{4}$ or more only half the time. [Why?] Yet when we solved the original recurrence, we assumed that c really is a constant across all calls. So we cannot make direct use of $E[C]$ to show that the recurrence has a linear solution.

To see where things go wrong if we try to follow our noses through the math, here's what happens if we just plug in the expected value for C into our recurrence and apply expectations throughout. We obtain

$$\begin{aligned}
E[T(n)] &\leq E[an + T(Cn)] = an + E[T(Cn)] \\
&\overset{??}{=} an + T(E[C] \cdot n) = an + T(\frac{3}{4}n) \leq \left(\frac{1}{1-\frac{3}{4}}\right) = 4an = O(n)
\end{aligned}$$

This looks great, but it is actually bogus reasoning: the questionable equality (marked ??) doesn't necessarily hold. That is, it is not in general the case that $E[f(X)] = f(E[X])$— although this *is* true if f is a linear function of X. (Why? Because "the expectation of the sum is the sum of the expectations," so it follows that $E[bX] = bE[X]$ for any constant b.) But in this situation, linearity of f is what we're trying to prove in the first place! So our reasoning is circular, and this approach is a dead end.

Re-grouping the recurrence. Here is another approach that *does* work. While the reasoning is more complicated, the math is simpler. (This is from the lecture notes for `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012`).

First, notice that although the probability of shrinking the problem by $\frac{3}{4}$ on a given call is only $\frac{1}{2}$, the probability of shrinking it by $\frac{3}{4}$ over *two* successive calls is much higher, namely $(1-\frac{1}{2})^2 = \frac{1}{4}$. Why? Because to avoid shrinking by $\frac{3}{4}$ over two calls, we have to be unlucky twice. On each call we are lucky half the time, hence unlucky $(1-\frac{1}{2})$ the time. Moreover, our chances of being unlucky on the first call are *independent* of our chances on the second call, so the overall probability is the *product* of the probabilities. Generalizing, we can see that the probability of shrinking by $\frac{3}{4}$ after s calls is $1-(\frac{1}{2})^s$, which rapidly approaches 1 as s grows.

This suggests a different analytical strategy. Rather than worrying about how much the problem shrinks on each recursive call, let's ask instead how many levels of recursive calls are needed before the problem has shrunk by a *fixed* ratio. The exact ratio we choose doesn't matter as long as it is between $\frac{1}{2}$ and 1, so let's continue using $\frac{3}{4}$. Of course, we don't know exactly how many levels are needed, so we need to define some random variables. Let N_0 to be the number of recursive calls that occur until just before the problem size drops to below $\frac{3}{4}$ of the initial size n. More generally, let N_r be the number of recursive calls that occur after the problem size has dropped below $(\frac{3}{4})^r n$, but before it has dropped below $(\frac{3}{4})^{r+1} n$. For example, if it takes five recursive calls to shrink the problem down to size $\frac{3}{4}n$, and eight recursive calls to shrink it down to size $(\frac{3}{4})^2 n$, then $T_0 = 5$ and $T_1 = 3$.

Now observe that we can unfold the recurrence for $T(n)$ and then "batch together" the terms corresponding to the first N_0 recursive calls, then to the next N_1 recursive calls, and so on:

$$\begin{aligned} T(n) =& an+ \\ & ac_1 n + ac_2 n + \cdots + ac_{N_0} n + \\ & ac_{N_0+1} n + ac_{N_0+2} n + \cdots + ac_{N_o+N_1} n + \\ & \cdots \end{aligned}$$

where each c_i is the (unknown) shrinkage factor corresponding to the ith recursive call. Consider the second line of this recurrence. By the definition of N_0, all the shrinkage factors $c_1, \ldots c_{N_0}$ are between $\frac{3}{4}$ and 1, so the sum of the terms on this line is bounded above by $an \cdot N_0$. Similarly, on the third line, all the shrinkage factors $c_{N_0+1}, \ldots, c_{N_o+N_1}$ are between $(\frac{3}{4})^2$ and $\frac{3}{4}$, so the sum of the terms on this line is bounded above by $a(3n/4) \cdot N_1$. Generalizing and going to the limit, we get an overall bound

$$T(n) \le \sum_{r=0}^{\infty} a(\frac{3}{4})^r n \cdot N_r$$

Taking expectations and using the "expectation of the sum" rule, we get

$$E[T(n)] \le \sum_{r=0}^{\infty} a(\frac{3}{4})^r n \cdot E[N_r]$$

Now it just remains to determine the $E[N_r]$. From our previous discussion we have that

$$\Pr\{N_r > s\} = (\frac{1}{2})^s$$

from which it is straightforward to calculate

$$E[N_r] \le \sum_{s=0}^{\infty} s \cdot \Pr\{N_r > s\} = \sum_{s=0}^{\infty} s \cdot (\frac{1}{2})^s = \frac{(1/2)}{(1-(1/2))^2} = 2$$

where the next-to-last equality follows from CLRS equation (A.8). Crucially, this expression does not depend on r. In other words, on average it takes two recursive calls to shrink the problem size by $\frac{3}{4}$ at *any* point in the recursive call sequence.

Plugging this back into the expectation equation above, we get

$$E[T(n)] \le \sum_{r=0}^{\infty} 2a(\frac{3}{4})^r n$$

which is just a closed form for

$$E[T(n)] \le 2an + E[T(3n/4)]$$

which, in turn, is just an instance of our original recurrence with $2a$ in place of a and $c = \frac{3}{4}$. So finally, we indeed have $E[T(n)] = O(n)$, as desired.