Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
October 25th, 2024

2.3: Quicksort

Exercise 1. Show, in the style of the trace given with `partition()`, how that method partitions the array `E A S Y Q U E S T I O N`.

Solution. We set `lo` to `0` which means `E` is the partition key. Then we start with `i = lo` and `j = 12` (this is `hi = 11` plus 1). We use the `i` index to scan from the left, starting with `++i` (and hence `lo + 1`) and comparing it against `a[lo]` which is `E`. If we encounter something equal to or larger than `E`, we stop. Similarly, we scan from the right with index j, starting with `--j` (meaning `hi - 1` or `11` is the first index) and then continue until we encounter a key smaller or equal to the partition key `E`.

			a[]											
	i	j	0	1	2	3	4	5	6	7	8	9	10	11
initial values	0	12	E	A	S	Y	Q	U	E	S	T	I	O	N
scan left, scan right	2	6	E	A	S	Y	Q	U	E	S	T	I	O	N
exchange	2	6	E	A	E	Y	Q	U	S	S	T	I	O	N
scan left, scan right	3	2	E	A	E ←	Y →	Q ←	U ←	S	S	T	I	O	N
final exchange	3	2	E	A	E	Y	Q	U	S	S	T	I	O	N
result		2	E	A	E	Y	Q	U	S	S	T	I	O	N

Exercise 2. Show, in the style of the quicksort trace given in this section, how quicksort sorts the array `E A S Y Q U E S T I O N` (for the purposes of this exercise, ignore the initial shuffle).

Solution.

			a[]											
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11
			E	A	S	Y	Q	U	E	S	T	I	O	N
0	2	11	E	A	E	Y	Q	U	S	S	T	I	O	N
0	1	1	A	E	E	Y	Q	U	S	S	T	I	O	N
0		0	A	E	E	Y	Q	U	S	S	T	I	O	N
3	11	11	A	E	E	N	Q	U	S	S	T	I	O	Y
3	4	10	A	E	E	I	N	U	S	S	T	Q	O	Y
3		3	A	E	E	I	N	U	S	S	T	Q	O	Y
5	10	10	A	E	E	I	N	O	S	S	T	Q	U	Y
5	5	9	A	E	E	I	N	O	S	S	T	Q	U	Y
6	7	9	A	E	E	I	N	O	Q	S	T	S	U	Y
6		6	A	E	E	I	N	O	Q	S	T	S	U	Y
8	9	9	A	E	E	I	N	O	Q	S	S	T	U	Y
8		8	A	E	E	I	N	O	Q	S	S	T	U	Y
			A	E	E	I	N	O	Q	S	S	T	U	Y

Exercise 3. What is the maximum number of times during the execution of `Quick.sort()` that that the largest item can be exchanged, for an array of length `n`?

Solution. Suppose that the array's entries are all distinct. If the largest key is at the end, then no exchange will ever occur:

```
* * * * * L
```

Here, `L` stands for largest. If the largest key is at the beginning, then it will be the first partition key, and one exchange will occur.

```
L * * * * *
* * * * * L
```

Thus, at least one exchange occurs. To explore whether more than one can occur, we can consider the case when it is not the partition item, and not at the last position.

If the partition item is not the largest key, then right scan will never end due to a comparison with the largest largest key. This is because the right scan continues as long as a key larger than the partition key is encountered. One consequence is that the largest item will not be involved in the final exchange, because `j` will always move past the largest key if it encounters it. Therefore, all exchanges with the largest item occur when the right scan encounters the largest key and the scan indices have not crossed. In that case, since it is encountered by the right scan, any swap exchanging it with the key encountered on the left scan will move the largest key forward. That is, the largest key is never moved to a lower position. Since we're assuming that it is not the partitioning key and not at the end, this suggest there's at most $n-2$ moves forward.

Though $n-2$ is an upper bound on the number of exchanges, I don't know if it is the maximum because I cannot think of a distribution of the keys that would cause all $n-2$ exchanges to occur (or for that matter, a scenario where more than 2 exchanges occur). For example, it could be 2 times if it's to the right of the partitioning key, and the value at the end of the array is larger than the partitioning key.

```
2 L * * 1 3
2 1 * * L 3
```

In that case, the largest key is exchanged up, but not to the last position, so that at least one more exchange will be at a different partitioning stage in order to place it at the end. See user named Panic on StackOverflow who claims the maximum is $\lfloor n/2 \rfloor$ and gives an example for $n = 10$.

Exercise 4. Suppose that the initial random shuffle is omitted. Give six arrays of ten elements for which `Quick.sort()` uses the worst-case number of compares?

Solution.

```
// sorted array, distinct keys
1 2 3 4 5 6 7 8 9 10

// inversely sorted, distinct keys
10 9 8 7 6 5 4 3 2 1

// largest followed by increasing sequence
```

```
10 1 2 3 4 5 6 7 8 9

// smallest followed by decreasing sequence
1 10 9 8 7 6 5 4 3 2

// bitonic (increase then decrease)
1 2 3 4 5 10 9 8 7 6
```

Exercise 5. Give a code fragment that sorts an array that is known to consist of items having just two distinct keys.

Solution. The 3-way partitioning method would work for this. However, I came up with the following:

```
// Find first index of largest key.
int i = 0;
while (i < a.length - 1 && !less(a[i + 1], a[i]))
   i++;
for (int j = i + 1; j < a.length; j++) {
   if (less(a[j], a[j - 1]))
      exch(a, i++, j);
}
```

Exercise 6. Write a program to compute the exact value of C_n, and compare the exact value with the approximation $2n \ln n$, for $n = 100, 1,000$, and $10,000$.

Solution. See the `com.segarciat.algs4.ch2.sec3.ex06.QuicksortCompares` class.

Exercise 7. Find the expected number of sub-arrays of size 0, 1, and 2 when quicksort is used to sort an array of N items with distinct keys. If you are mathematically inclined, do the math; if not, run experiments to develop hypotheses.

Solution. One idea I had was to call $T(N)$ the expected number of arrays of size 0, 1, and 2 for an array of N distinct items. We know that $T(0) = 1$, since the array has size 0 and quicksort will not be called recursively to partition it. Similarly, $T(1) = 1$. For $T(2)$, if we consider the original array of size 2 to be sub-array, then at $T(2) \geq 1$. When quicksort encounters such an array, it will always partition it into an array of size 0 and an array of size 1. Thus, $T(2) = 3$.

Now consider the case where $N \geq 3$. We know `Quick.sort()` partitions the array at each step. For example, if $j = 0$ is the first index chosen after `partition()` is called, then this leads to recursively calls to sort `a[0..-1]` and `a[1..N-1]`, which are arrays of size 0 and size $N - 1$, respectively. Similarly, if $j = 1$ is chosen as the first partition index instead, then the array is split into `a[0..0]` and `a[2..N-1]`, which are arrays of size 1 and $N - 2$, respectively. Continuing this way, for $N \geq 3$, we can express $T(N)$ as

a recurrence by taking the average as we break it up as follows:

$$\begin{aligned}
T(0) &= 1 \\
T(1) &= 1 \\
T(2) &= 3 \\
T(N) &= \frac{1}{N}\left([T(0)+T(N-1)]+[T(1)+T(N-2)]+\cdots+[T(N-1)+T(0)]\right) \\
&= \frac{1}{N}\sum_{j=0}^{N-1}[T(j)+T(N-j)] \\
&= \frac{2}{N}\cdot\sum_{j=0}^{N-1}T(j), \quad N \geq 3
\end{aligned}$$

Then, I followed Paul Sinclair's suggestion on this StackExchange post and defined $S(N) = \sum_{j=0}^{N} T(j)$ in order to transfer my recurrence into one that would be easier to solve. For $N \geq 3$, this meant that

$$S(N) - S(N-1) = \frac{2}{N}S(N-1), \quad N \geq 3$$

Thus, the recurrence for $S(N)$ is

$$\begin{aligned}
S(0) &= 1 \\
S(1) &= 2 \\
S(2) &= 5 \\
S(N) &= \frac{N+2}{N}S(N-1), \quad N \geq 3
\end{aligned}$$

I did not know how to solve this recurrence but according to Wolframalpha, the solution turns out to be

$$S(N) = \frac{1}{2}\cdot c\cdot(N+1)(N+2), \quad N \geq 3.$$

By the recurrence relation for $S(N)$, we find that

$$S(3) = \frac{3+2}{3}S(2) = \frac{5}{3}\cdot 5 = \frac{25}{3}$$

Thus, we can solve for c:

$$\frac{25}{3} = S(3) = \frac{1}{2}c\cdot(3+1)(3+2) = \frac{c}{2}\cdot(4)(5) = 10c$$

so

$$c = \frac{25}{30} = \frac{5}{6}.$$

Now for $N \geq 4$, we can use the recurrence for $S(N)$ to get

$$\begin{aligned} T(N) &= S(N) - S(N-1) \\ &= \frac{2}{N} S(N-1) \\ &= \frac{2}{N} \cdot \frac{1}{2} \cdot c \cdot N(N+1) \\ &= c(N+1) \\ &= \frac{5}{6}(N+1) \end{aligned}$$

Thus, $T(N)$ is proportional to the N, the size of the array, and the constant of proportionality is $\frac{5}{6}$. Thus, quicksort will produce about $\sim \frac{5}{6}N$ sub-arrays of size 0, 1, and 2, on the average.

See `com.segarciat.algs4.ch2.sec3.ex07.QuicksortCompares`, a program I wrote to count the number of arrays of size 2 or less and how it compares to the array size. Below is a sample output from a particular run:

```
n = 2, Count of Arrays of size 2 or less = 3, Ratio of count to n = 1.50
n = 4, Count of Arrays of size 2 or less = 4, Ratio of count to n = 1.00
n = 8, Count of Arrays of size 2 or less = 8, Ratio of count to n = 0.98
n = 16, Count of Arrays of size 2 or less = 14, Ratio of count to n = 0.89
n = 32, Count of Arrays of size 2 or less = 28, Ratio of count to n = 0.87
n = 64, Count of Arrays of size 2 or less = 53, Ratio of count to n = 0.83
n = 128, Count of Arrays of size 2 or less = 105, Ratio of count to n = 0.82
n = 256, Count of Arrays of size 2 or less = 213, Ratio of count to n = 0.83
n = 512, Count of Arrays of size 2 or less = 422, Ratio of count to n = 0.82
n = 1024, Count of Arrays of size 2 or less = 859, Ratio of count to n = 0.84
n = 2048, Count of Arrays of size 2 or less = 1707, Ratio of count to n = 0.83
n = 4096, Count of Arrays of size 2 or less = 3404, Ratio of count to n = 0.83
n = 8192, Count of Arrays of size 2 or less = 6845, Ratio of count to n = 0.84
n = 16384, Count of Arrays of size 2 or less = 13676, Ratio of count to n =
    0.83
n = 32768, Count of Arrays of size 2 or less = 27291, Ratio of count to n =
    0.83
n = 65536, Count of Arrays of size 2 or less = 54660, Ratio of count to n =
    0.83
n = 131072, Count of Arrays of size 2 or less = 109285, Ratio of count to n =
    0.83
n = 262144, Count of Arrays of size 2 or less = 218374, Ratio of count to n =
    0.83
n = 524288, Count of Arrays of size 2 or less = 436873, Ratio of count to n =
    0.83
n = 1048576, Count of Arrays of size 2 or less = 873932, Ratio of count to n
    = 0.83
n = 2097152, Count of Arrays of size 2 or less = 1747220, Ratio of count to n
    = 0.83
```

These results support the mathematical analysis because $\frac{5}{6} = 0.8\overline{3}$.

Exercise 8. About how many compares will `Quick.sort()` make when sorting an array of n items that are all equal?

Solution. Consider partition call. Since the partition key is always equal to the item it is compared against, the left and right scans always move by 1 before a swap is necessary. As a result, the number of times that the left scan index increases and the number of times the right scan index decreases are within 1 of one another (depending the parity of n). Thus, the scan indices cross around the center of the array, and partition index `j` falls around the middle. At this point, about $\sim n$ compares have occurred, once for each compare against the left pointer and one for each compare against the right pointer. Since all keys are equal, the process now proceeds by induction when it is cut in half, now yielding about $n/2$ compares for each half. If we see it as a binary tree, then its height is about $\sim \lg n$ and at each level there's about n compares. Thus we get $\sim n \lg n$ compares overall.

Exercise 9. Explain what happens when `Quick.sort()` is run on an array having items with just two distinct keys, and then explain what happens when it is run on an array having just three distinct keys.

Solution. First consider the case with two distinct keys (for simplicity, say their values are 1 and 2). Suppose the smallest key is at the beginning, meaning a 1. The right scan index `i` will stop at every comparison. The left scan index `j` will stop each time a 1 is encountered, causing a swap to occur. By the time the scan indices cross, the array will sorted. However, the sorting algorithm will continue, and it will now work on two halves that both consist of only equal items. The sort the continues as in Exercise 8. A similar case occurs when sorting the pivot is the largest item.

When it consists of only three distinct items, call them 1, 2, 3. Then there's a few cases:

- If the first pivot is 1, then once again the left scan index always stops, and so does the right one. The end result of this first partition `a[lo..j]` has all the 1s, and`a[j+1..hi]` has all the 2s and 3s. Now this continues as in the 2 item case.
- If the first pivot is 3, then the right scan index stops only when a 3 is encountered. The left scan index stops every time. By the end, `a[j..hi]` has all the 3s, and `a[lo..j-1]` has the 1s and 2s. Now we proceed as in the 2 item case.
- If the first pivot is 2, then by the end, `a[lo..j-1]` has 1s and 2s, and `a[j+1..hi]` has 2s and 3s. Thus both cases proceed as int he 2 element case.

Exercise 10. *Chebyshev's inequality* says that the probability that a random variable is more than k standard deviations away from the mean is less than $1/k^2$. For $n = 1$ million, use Chebyshev's inequality to bound the probability that the number of compares used by quicksort is more than 100 billion ($0.1n^2$).

Solution. Let X be a random variable whose values correspond to the number of compares used by quicksort when sorting an array of $n = 1$ million values. We want to obtain a bound for the probability that $X \geq 100$ billion.

By Proposition K in [SW11], quicksort uses $\sim 2n \ln n$ compares on the average. Since $n = 10^6$, this means the average is about $\mu = 161,180,956$, which is about 162 million. Thus we must determined how many standard deviations 100 billion is from the mean. According to the proof of Proposition L, the standard deviation of the number of compare

is about $0.65n$. Therefore, by subtracting 1 million from the mean and dividing by $0.65n$, we get the number of standard deviations that 100 billion is from the mean. Thus

$$\begin{aligned} k &\approx \frac{0.1n^2 - \mu}{\sigma} \\ &= \frac{0.1n^2 - 2n\log n}{0.65n} \\ &= \frac{0.1n - 2\log n}{0.65} \\ &\approx 153803 \end{aligned}$$

By Chebyshev's inequality, the probability that $X \geq 10^{11}$ (100 billion) is less than $\frac{1}{k^2} \approx 4.23 \times 10^{-11}$.

Exercise 11. Suppose that we can over items with keys equal to the partitioning item's key instead of stopping the scans when we encounter them. Show that the running time of this version of quicksort is quadratic for all arrays with just a constant number of distinct keys.

Solution.

Proof. Suppose quicksort is called on an array of n items, with k distinct keys, and k is a constant independent of the array size n. Here are assume $k \ll n$ (meaning k is much less than n).

Suppose first $k = 1$ so that all keys are equal. When quicksort is called on such an array, the fact that all keys are equal and that we scan over items with equal to the partition key means that that the index `i` that scans to the right will reach `hi`, the right end of the sub-array. Similarly, the `j` index scanning towards the left will reach `lo`. Since the indices cross, the loop ends, and the pivot is exchanged with itself. That result is that `lo` becomes the partitioning index, and the method is called recursively with an array of size 0 and one of size `hi - lo`. Put another way, the sub-array only decreases by one in size and we perform the full amount of comparisons. This means the number of comparisons is

$$2n + 2(n-1) + \cdots + 2 = 2 \cdot \frac{(n+1)n}{2} = (n+1)n$$

Thus the algorithm is quadratic for $k = 1$. Now suppose that algorithm is quadratic for $k \geq 1$, and that there are $k+1$ distinct keys. If the item at index 0 is the smallest, then the first call results in a partition of index of 0 because the `j` index scanning from right to left is less than every key except itself, and skips keys equal to itself. Thus it reaches index 0. This continues happening until the value of `lo` is not the smallest key. A similar thing happens if the item at index 0 is the largest key. In both cases, the number of comparisons is proportional to the size of the sub-array and the sub-array size only decreases by 1 because there's an empty sub-array at each step.

When this the pivot is no longer the smallest nor the largest element, the sub-array is partitioned into two parts, both of which have less than $k+1$ distinct keys. By induction, this takes quadratic time. □

Exercise 12. Show, in the style of the trace given with the code, how the 3-way quicksort first partitions the array `B A B A B A B A C A D A B R A`.

Solution.

lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	14	B	A	B	A	B	A	B	A	C	A	D	A	B	R	A
0	1	14	B	A	B	A	B	A	B	A	C	A	D	A	B	R	A
1	2	14	A	B	B	A	B	A	B	A	C	A	D	A	B	R	A
1	3	14	A	B	B	A	B	A	B	A	C	A	D	A	B	R	A
2	4	14	A	A	B	B	B	A	B	A	C	A	D	A	B	R	A
2	5	14	A	A	B	B	B	A	B	A	C	A	D	A	B	R	A
3	6	14	A	A	A	B	B	B	B	A	C	A	D	A	B	R	A
3	7	14	A	A	A	B	B	B	B	A	C	A	D	A	B	R	A
4	8	14	A	A	A	A	B	B	B	B	C	A	D	A	B	R	A
4	8	13	A	A	A	A	B	B	B	B	A	A	D	A	B	R	C
5	9	13	A	A	A	A	A	B	B	B	B	A	D	A	B	R	C
6	10	13	A	A	A	A	A	A	B	B	B	B	D	A	B	R	C
6	10	12	A	A	A	A	A	A	B	B	B	B	R	A	B	D	C
6	10	11	A	A	A	A	A	A	B	B	B	B	B	A	R	D	C
6	11	11	A	A	A	A	A	A	B	B	B	B	B	A	R	D	C
7	12	11	A	A	A	A	A	A	A	B	B	B	B	B	R	D	C
7	12	11	A	A	A	A	A	A	A	B	B	B	B	B	R	D	C

Exercise 13. What is the *recursive depth* of quicksort, in the best, worst, and average cases? This is the size of the stack that the system needs to keep track of the recursive calls. See Exercise 2.3.20 for a way to guarantee that the recursive depth is logarithmic in the worst case.

Solution. In the worst case, where one array is empty for each recursive call after `partition()`, the depth is $n - 1$. In the best case, the array is divided into roughly halves each time, so the stack size would be about $\sim \lg n$ in that case. Since the average case is closer to the best case than the worst case, the stack size is about $\sim \lg n$ in the average as well.

Exercise 14. Prove that when running quicksort on an array with n distinct items, the probability of comparing the ith and jth smallest items is $2/(j - i + 1)$. Then use this result to prove Proposition K in [SW11].

Solution.

Proof. Assume $j > i$. Then $j - i + 1$ is the number of integer indices in range `i..j`. Since the `partition()` algorithm only compares the items against the current pivot, meaning the current value of `a[lo]`, it follows that the `i`th and `j`th smallest items are only compared if either one is a pivot of the current sub-array. Moreover, they can only be compared if they belong to the same sub-array. This means that if the `k`th smallest element is chosen as a partition key before both `i` and `j`, for k in the open interval (i, j), then the `i`th and `j`th smallest element will not be compared.

Let E be the event that the `i`th and `j`th smallest elements are compared, and let F be the event that the first item from the sorted subsequence belonging to the closed interval `[i, j]` that is chosen to partition the array is either `i` or `j`. In particular, if F does not occur, then the `k`th element chosen from the sorted subsequence `[i, j]` satisfies

$i < k < j$, so `i` and `j` will not be compared. That is, E will not occur. It follows that this conditional probability is 0: $\mathrm{P}(E|F^{\mathsf{c}}) = 0$. Then by the Law of Total Probability, we have

$$\begin{aligned}\mathrm{P}(E) &= \mathrm{P}(E|F)\mathrm{P}(F) + \mathrm{P}(E|F^{\mathsf{c}})P(F^{\mathsf{c}}) \\ &= \mathrm{P}(E|F)\mathrm{P}(F)\end{aligned}$$

Since every element in `[i, j]` is equally likely to be chosen, we find that

$$\mathrm{P}(F) = \frac{|\{\texttt{i, j}\}|}{|\texttt{[i, j]} \cap \mathbb{N}|} = \frac{2}{j-i+1}$$

If F occurs, then E will certainly occur, so $\mathrm{P}(E|F) = 1$, and hence, $\mathrm{P}(E) = \mathrm{P}(F) = 2/(j-i+1)$.

Now to prove Proposition K, let $\overline{x}$ be the average number of compares done by quicksort. Note that once the ith and jth element are compared, they will never be compared again. This is because they are compared only when either one is the partition element, and beyond that, that element will be fixed and no more comparisons are done against it. Hence, since the average is the sum of the frequency of each event multiplied by the probability of the event, we have

$$\begin{aligned}\overline{x} &= \sum_{j>i} \mathrm{P}(i\text{th and } j\text{th smallest compared}) \cdot (\#\text{ times } i\text{th and } j\text{th smallest compared}) \\ &= \sum_{j=1}^{n}\sum_{i=1}^{j-1} \frac{2}{j-i+1} \\ &\approx \sum_{j=1}^{n} \int_1^j \frac{2}{j-x+1}\,dx \\ &= \sum_{j=1}^{n} [-2\ln|j-x+1|]_{x=1}^{x=j} \\ &= 2\cdot\sum_{j=1}^{n}(\ln|j| - \ln|1|) \\ &= 2\cdot(\ln 1 + \ln 2 + \cdots + \ln n) \\ &= 2\cdot\ln(n!) \\ &\approx 2n\ln n\end{aligned}$$

The first approximation (with the integral) made use of the interpretation of the summation as a finite Riemann sum, which serves as an approximation for the integral. The last approximation is Stirling's approximation. □

Exercise 15. *Nuts and bolts.* (G. J. E. Rawlins). You have a mixed pile of n nuts and n bolts and need to quickly find the corresponding pair of nuts and bolts. Each nut matches exactly one bolt, and each bolt matches exactly one nut. By fitting a nut and bolt together, you can see which is bigger, but it is not possible to directly compare two nuts or two bolts. Give an efficient method for solving the problem.

Solution. Start with a nut. Take any bolt. If the bolt is too big for the nut, place it to the right. If the nut is too small for nut, place it to the left. Otherwise, place it in the center. Now all bolts will be on the table, and we've also found the bolt matching the starting nut. Before discarding the match, we pick up another nut, and compare it against the matched bolt. If the bolt is too big, than the bolt for the new nut is on the left bunch of bolts. Otherwise, the bolt for the new nut is on the right. We discard the matched nut and bolt and move on to the correct side. We continue this way until we've found all of the matches. Each time we do this, we pair a nut with a bolt, so the method solves the problem. The fact that it is efficient follows from noticing that this is just applying quicksort.

Exercise 16. *Best case.* Write a program that produces a best-case array (with no duplicates) for `sort()` in Algorithm 2.5: an array of n items with distinct keys having the property that every partition will produce sub-arrays that differ in size by at most 1 (the same sub-array lengths that would happen for n equal keys). (For the purpose of this exercise, ignore the initial shuffle).

Solution. In order to obtain the best case, we need an array where the partition key is always the median (or close to it) of the current sub-array.

To see this concretely, say $n = 16$, and that we allow the possible keys to be 0 through $n-1 = 15$, which are also (intentionally) the valid index values for an array of size $n = 16$.

The first median is 7, and we can compute it as

$$\left\lfloor 0 + \frac{15 - 0}{2} \right\rfloor = 7$$

We want it to be the first partition key, so we set `a[0] = 7`. Say we call quicksort on `a` with 7 as `a[0]`. Since the `partition()` method always performs the final exchange with `j`, the last item in the left sub-array (with keys smaller than 7), that element becomes the next partition key for the left sub-array. By the same reasoning, we want it to be the median, and the median of `0..6` is 3, so it should be 3. Now we have something like this so far:

```
7 < < < < < < 3 > > > > > > > >
```

The `<` and `>` are placeholders for items that are less than greater than the partition key (7), before the final exchange (recall the items are all distinct). Notice that we have here `a[7] = 3`. What about the partition key for the right sub-array for keys `8..15`? The partition key for the right sub-array will be 8, so it should be the median of the right sub-array, which is 11. So now we have:

```
7 < < < < < < 3 11 > > > > > > >
```

In this case, by knowing the median 7, we computed median of the left sub-array and assigned it to `a[7]`, then the median of the right sub-array and assigned it to `a[7+1]`.

To understand how we would proceed, let's exchange 7 and 3 as quicksort would do, and consider what remains:

```
3 * * * * * * 7 11 * * * * * * * // after swap
3 * * * * * * // focus us now lower sub-array
```

As before, when `partition()` is called on this sub-array, the final exchange that occurs will involve the next partition item. After the exchange, all items on the left sub-array will be less than 3. The next partitioning item of the left sub-array should be the median of the next left sub-array. The left sub-array would have `0..2`, and its median is 1, so the next partition index should be 1:

```
3 < < 1 > > >
```

The right sub-array `4..6` has a median of `5`, which becomes its next partition key:

```
3 < < 1 5 > >
```

Thus we see that we want `a[3] = 1` and `a[3 + 1] = 5`. If we exchange 3 and 1 as would `partition()` in quicksort, so as to continue:

```
1 * * 3 5 * * // after swap
1 * * // focus on lower sub-array
```

Now similar to before, we consider the median of the items left of the partition key (1). The left sub-array is just `0..0` and the right sub-array is `2..2`, so we conclude that we should have:

```
1 0 2
```

So we set `a[1] = 0` and `a[1 + 1] = 2`. Similarly for the sub-array with `5` as its partition key we have:

```
5 4 6
```

If we continued this process, we would get:

```
7 0 2 1 5 4 6 3 11 8 10 9 13 12 14 15
```

At each step, the middle value served as the index for the median of the lower half, and the middle value plus 1 served as the index of the upper half. Then to proceed, the algorithm continued in the lower half and upper half going left first, in a recursive way.

To formalize this algorithm, start with `lo = 0`, `hi = n - 1`:

(i) *Compute the median as*: `mid = lo + (hi - lo) / 2`.

(ii) *Set median of left sub-array*: `a[mid] = median(a, lo, mid-1)`. This is what `a[lo]` will first exchange with in quicksort, and it becomes the next partition key of the left sub-array.

(iii) *Set median of right sub-array*: `a[mid+1] = median(a, mid+1, hi)`. This is what becomes the partition key of the right sub-array.

The algorithm eventually ends because the search space is cut roughly in half each time. The median of an empty range such as `lo..lo` is precisely `lo`, so this becomes the base case. We begin the chain of recursive calls with `a[0] = median(a, 0, n - 1)`, which would set `a[0] = 7` in our example above with $n = 16$.

Below is a trace of this for $n = 16$. A green on the left columns under the headers `lo` and `mid` denotes a return value as we ascend back up in recursion depth. It is the next

value that will be placed in the array. A red index under `mid` or `mid + 1` is the index where that return value will be placed. On the right side under the array indices, a star * denotes an entry that has yet to be placed, a gray entry is one that has already been assigned, and a green entry is one that is assigned in the current step.

lo	mid	mid + 1	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0			15	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	7	8	15	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	3	4	6	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	1	2	2	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0			0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	1	2	2	*	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*
2			2	*	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	1	2	2	*	0	2	*	*	*	*	*	*	*	*	*	*	*	*	*
0	3	4	6	*	0	2	1	*	*	*	*	*	*	*	*	*	*	*	*
4	5	6	6	*	0	2	1	*	*	*	*	*	*	*	*	*	*	*	*
4			4	*	0	2	1	*	*	*	*	*	*	*	*	*	*	*	*
4	5	6	6	*	0	2	1	*	4	*	*	*	*	*	*	*	*	*	*
6			6	*	0	2	1	*	4	*	*	*	*	*	*	*	*	*	*
4	5	6	6	*	0	2	1	*	4	6	*	*	*	*	*	*	*	*	*
0	3	4	6	*	0	2	1	5	4	6	*	*	*	*	*	*	*	*	*
0	7	8	15	*	0	2	1	5	4	6	3	*	*	*	*	*	*	*	*
8	11	12	15	*	0	2	1	5	4	6	3	*	*	*	*	*	*	*	*
8	9	10	10	*	0	2	1	5	4	6	3	*	*	*	*	*	*	*	*
8			8	*	0	2	1	5	4	6	3	*	*	*	*	*	*	*	*
8	9	10	10	*	0	2	1	5	4	6	3	*	8	*	*	*	*	*	*
10			10	*	0	2	1	5	4	6	3	*	8	*	*	*	*	*	*
8	9	10	10	*	0	2	1	5	4	6	3	*	8	10	*	*	*	*	*
8	11	12	15	*	0	2	1	5	4	6	3	*	8	10	9	*	*	*	*
12	13	14	15	*	0	2	1	5	4	6	3	*	8	10	9	*	*	*	*
12			12	*	0	2	1	5	4	6	3	*	8	10	9	*	*	*	*
12	13	14	15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	*	*
14	14	15	15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	*	*
14			13	*	0	2	1	5	4	6	3	*	8	10	9	*	12	*	*
14	14	15	15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	14	*
15			15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	14	*
14	14	15	15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	14	15
12	13	14	15	*	0	2	1	5	4	6	3	*	8	10	9	*	12	14	15
8	11	12	15	*	0	2	1	5	4	6	3	*	8	10	9	13	12	14	15
0	7	8	15	*	0	2	1	5	4	6	3	11	8	10	9	13	12	14	15
0			15	7	0	2	1	5	4	6	3	11	8	10	9	13	12	14	15
				7	0	2	1	5	4	6	3	11	8	10	9	13	12	14	15

See `com.segarciat.algs4.ch2.sec3.ex16.QuicksortBestCase` for my implementation of this algorithm. To verify the effectiveness of this algorithm, I used my program from Exercise 2.3.6 to compare the actual compares to the average number of compares, and computed ratios of these quantities. I modified the quicksort algorithm to not perform the random shuffle to ensure the "best-case" array produced by my algorithm was used as-is. Below are the results of a sample run for different array sizes:

```
n=2, actual=3, average=2, actualToAverageRatio=1.50
n=4, actual=8, average=8, actualToAverageRatio=1.02
n=8, actual=21, average=24, actualToAverageRatio=0.88
n=16, actual=54, average=66, actualToAverageRatio=0.82
n=32, actual=135, average=171, actualToAverageRatio=0.79
n=64, actual=328, average=424, actualToAverageRatio=0.77
n=128, actual=777, average=1017, actualToAverageRatio=0.76
n=256, actual=1802, average=2379, actualToAverageRatio=0.76
n=512, actual=4107, average=5457, actualToAverageRatio=0.75
n=1024, actual=9228, average=12321, actualToAverageRatio=0.75
n=2048, actual=20493, average=27467, actualToAverageRatio=0.75
n=4096, actual=45070, average=60597, actualToAverageRatio=0.74
n=8192, actual=98319, average=132535, actualToAverageRatio=0.74
n=16384, actual=213008, average=287765, actualToAverageRatio=0.74
n=32768, actual=458769, average=620938, actualToAverageRatio=0.74
n=65536, actual=983058, average=1332707, actualToAverageRatio=0.74
n=131072, actual=2097171, average=2847097, actualToAverageRatio=0.74
n=262144, actual=4456468, average=6057579, actualToAverageRatio=0.74
n=524288, actual=9437205, average=12841950, actualToAverageRatio=0.73
n=1048576, actual=19922966, average=27137510, actualToAverageRatio=0.73
n=2097152, actual=41943063, average=57182262, actualToAverageRatio=0.73
n=4194304, actual=88080408, average=120179035, actualToAverageRatio=0.73
n=8388608, actual=184549401, average=251987120, actualToAverageRatio=0.73
n=16777216, actual=385875994, average=527232369, actualToAverageRatio=0.73
n=33554432, actual=805306395, average=1100981024, actualToAverageRatio=0.73
```

The data suggests that in order for quicksort to sort the array produced by my algorithm, it requires about 75% of the compares that it would normally need on the average to sort a randomly ordered array. This supports the effectiveness of the algorithm in producing a "best-case" array.

Exercise 17. *Sentinels.* Modify the code in Algorithm 2.5 to remove both bounds checks in the inner `while` loops. The test against the left end of the subarray is redundant since the partitioning item acts as a sentinel (`v` is never less than `a[lo]`). To enable removal of the other test, put an item whose key is the largest in the whole array into `a[a.length-1]` just after the shuffle. This item will never move (except possibly to be swapped with an item having the same key) and will serve as a sentinel in all subarrays involving the end of the array. *Note*: For a subarray that does not involve the end of the array, the leftmost entry to its right serves as a sentinel for the right end of the subarray.

Solution. See `com.segarciat.algs4.ch2.sec3.ex17.QuickSentinel.sort()`.

Exercise 18. *Median-of-3-partitioning.* Add median-of-3 partitioning to quicksort, as described in the text (see page 296). Run doubling tests to determine the effectiveness of the change.

Solution. See `com.segarciat.algs4.ch2.sec3.ex18.QuickMedianOf3.sort()`. The improvement was in the order of about 10% or so.

Exercise 2.3.19. *Median-of-5 partitioning.* Implement a quicksort based on partitioning on the median of a random sample of five items from the sub-array. Put the items of

the sample at the appropriate ends so that only the median participates in partitioning. Run doubling tests to determine the effectiveness of the change, in comparison both to the standard algorithm and to the median-of-3 partitioning (see the previous exercise). *Extra credit*: Devise a median-of-5 algorithm that uses fewer than seven compares on any input.

Solution. My first mistake when attempting this problem was trying to sort the items. However, after some thought, I recalled that Proposition I in [SW11] says that no compare-based sorting algorithm can guarantee to sort n items with fewer than $\lg(n!)$ compares. With a sample of 5 keys, we have $\lg(5!) \approx 6.9$, so it's not possible to sort the items with fewer than 7 compares! Therefore I focused on finding the median of the 5, which should be the partition key, and a key no less than the median, which will be a sentinel at the right side. Nevertheless, I was unable to do it in less than 7 compares, and my approach required at most 7 compares.

See `com.segarciat.algs4.ch2.sec3.ex19.QuickMedianOf5`.

Exercise 20. *Non-recursive quicksort.* Implement a non-recursive version of quicksort based on a main loop where a sub-array is popped from a stack to be partitioned, and the resulting sub-arrays are pushed onto the stack. *Note*: Push the larger of the sub-arrays onto the stack first, which guarantees that the stack will have at most $\lg n$ entries.

Solution. See `com.segarciat.algs4.ch2.sec3.ex20.NonRecQuick`.

Exercise 21. *Lower bound for sorting with equal keys.* Complete the first part of the proof of **Proposition M** in [SW11] by following the logic in the proof of **Proposition I** and using the observation that there are $n!/f_1!f_2!\cdots f_k!$ different ways to arrange keys with k different values, where the ith value appears with frequency f_i ($= np_i$, in the notation of **Proposition M**), with $f_1 + \cdots + f_k = n$.

Solution. The statement of **Proposition M** is the following:

> **Proposition** (Proposition M)**.** *No compare-based sorting algorithm can guarantee to sort n items with fewer than $nH - n$ compares, where H is the Shannon entropy, defined from the frequencies of key values.*

[SW11] defines the Shannon entropy as follows: if there are k distinct keys among the n elements of the array, and f_i is the frequency of each key, for integer i in $1 \leq i \leq k$, then

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \cdots + p_k \lg p_k)$$

where $p_i = f_i/n$, and $\sum_{i=1}^{n} f_i = n$.

Proof. In **Proposition I**, the idea was to use a binary tree to describe the sequence of compares that needed to be performed. Each internal node corresponded to a comparison between the ith and jth element. The leaves corresponded to a complete sort of the keys. Using the observation that there are $n!/f_1!f_2!\cdots f_k!$ to arrange keys with k different values, this quantity also represents the minimum number of leaves in such a tree. The other key fact in the proof was that the longest path from the root of the tree to a leaf represented the worst-case scenario for the number of compares for a given algorithm. The latter quantity represents the height h of the tree, and since a tree of height h

cannot have more leaves than a complete binary tree of height h (which has 2^h leaves), the inequality we arrive at is:

$$\frac{n!}{\prod_{i=1}^{k} f_i!} \leq \text{number of leaves} \leq 2^h$$

Then, using the fact that lg is monotone, applying lg on both sides yields

$$\lg\left(\frac{n!}{\prod_{i=1}^{k} f_i}\right) \leq \lg 2^h = h$$

To end the proof, we need to bound the value on the left of this inequality below. To prove this, we need to make use of Stirling's approximation:

$$\lg(n!) = n\lg n - n\lg e + O(\lg n)$$

Also, note that

$$\begin{aligned} \lg(f_i!) = \lg 1 + \cdots + \lg f_i \leq f_i \lg f_i \\ -\lg(f_i!) \geq -f_i \lg f_i \end{aligned}$$

Using the right-side of Stirling's approximation as a lower bound:

$$\begin{aligned} \lg\left(\frac{n!}{\prod_{i=1}^{k} f_i!}\right) &= \lg(n!) - \lg\left(\prod_{i=1}^{k} f_i\right) \\ &= \lg(n!) - \sum_{i=1}^{k} \lg(f_i!) \\ &\geq n\lg n - n\lg e - \sum_{i=1}^{k} f_i \lg f_i \\ &= n\lg n - n\lg e - n\sum_{i=1}^{k} p_i \lg(np_i) \\ &= n\lg n - n\lg e - n\sum_{i=1}^{k} p_i \lg n - n\sum_{i=1}^{k} p_i \lg p_i \\ &= n\lg n - n\lg e - n\lg n + nH \\ &= nH - n\lg e \end{aligned}$$

□

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.