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`.

	`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.

`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 $10,000$.

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.

References

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