Chapter 7
QuickSort

QuickSort is yet another example of the divide-and-conquer strategy.

To quickSort a subarray $A[p..r]$, when it contains more than one elements, we go through the following steps:

1) Pick up a pivot, $A[q]$, at position $q$.

2. Partition the array $A[p..r]$ into two subarrays $A[p..q-1]$ and $A[q+1..r]$, with one maybe empty, such that everything in the first subarray is no larger than (less than or equal to) the pivot; and everything in the second is larger.

3) Recursively sort the two subarrays.

We are done. 😊

An example might help...
A G-rated example

In the following, \( q \) is set to 4, and \( A[4]=5 \).

Notice the pivot 5 is in its final place (?), we then sort \( A[1..3] \) to \( [2, 3, 4] \); and \( A[6..7] \) to \( [6, 9] \), and we are done. 😊
A serious effort

Given the following code, which sorts elements sitting in the \([p, r]\) segment of \(A\),

\[
\text{QUICKSORT}(A, p, r)
\]

1. //If we have more than one elements
2. if \(p<r\)
3. //q gives where the pivot sits
4. then \(q<-\text{PARTITION}(A, p, r)\)
5. \(\text{QUICKSORT}(A, p, q-1)\)
6. \(\text{QUICKSORT}(A, q+1, r)\)

We then call \(\text{QUICKSORT}(A, 1, \text{length}[A])\) to sort \(A\), a list of anything that are comparable.

Clearly, the function \(\text{PARTITION}(A, p, r)\) is the only piece that we need to go through.

\textbf{Question:} How should we do \textit{it}?
The partition procedure

This procedure picks a *pivot*, $A[q]$, and rearranges the subarray $A[p..r]$ around this pivot.

PARTITION($A$, $p$, $r$)
1. $x \leftarrow A[r]$ //Use the last one as the pivot
2. $i \leftarrow p-1$
3. $j \leftarrow p$
5. while $j \leq r-1$
6. do if $A[j] \leq x$
7. then $i \leftarrow i+1$
8. exchange($A[i], A[j]$)
9. $j \leftarrow j+1$
10. //Put the pivot to the right place
11. exchange($A[i+1], A[r]$)
12. return $i+1$

**Questions:** 1. How does it work? Next page.

2. What happens when everything is the same (Exercise 7.1-2)? Hint: Look at the previous demo, and come up with a procedure.
I want to see...

Below shows how this partition procedure works...

Homework: Exercises 7.1-1 and 7.1-2.
What is going on?

If you play with it a bit, you will find out that the indices \( p, i, j, r \) essentially cut the array \( A \) into four pieces: \( A[p..i] \) contains all those elements less than or equal to the pivot; \( A[i+1, j-1] \) contains those elements larger than the pivot; \( A[j..r-1] \) contains those elements we are not sure yet; and \( A[r] \) is the pivot.

Hence, the loop invariant for the while loop in lines 5-9 is the following: At the beginning of each iteration of this for loop, for any index \( k \),

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 \( j \leq k \leq r - 1 \), then it is unknown.

4) If \( k = r \), then \( A[k] = x \).
The correctness

We now prove the correctness of the partition algorithm by checking the loop invariant.


Hence, the first two segments are empty, while the third contains everything, except $A[r]$, which is used as the pivot.

The base case holds.
The “inductive” part

Assume the property holds when the value of the loop variable $j$ is $j_0$, there are two cases to consider, depending on the outcome of the test $A[j_0] \leq x$.

If $A[j_0] > x$, we merely increment $j_0$ to include this element into the second segment. The invariant then holds at the beginning of the next iteration, when $j$ is increased to $j_0 + 1$. 
The other case

On the other hand, if $A[j_0] \leq x$, we now increment $i$ to $i_0 + 1$ and exchange $A[i_0]$, the first element in the second segment, and $A[j_0]$, which should belong to the first segment. After this swap, $A[i_0] \leq x$, and $A[j_0] > x$.

Together with the increment of $j$ to $j_0$ at the end of this loop, all the conditions of the invariant again hold.
Wrap it up

At the end, \( j = r \). Now, everything, except the pivot, has been categorized into either the first or the second segment.

We finally swap, in Lines 11 and 12, the pivot \( A[r] \) with the first element in the second segment \( A[i+1] \) to place the pivot into the “right place”, \( i+1 \), and return \( q=i+1 \).

Thus, \( A[p, r] \) now contains three pieces: \( A[p, q-1] \) contains everything that is no more than the pivot, \( A[q] \) contains the pivot; and \( A[q+1, r] \) contains everything bigger than the pivot.

*The important thing is that the pivot is in the right place.*
quickSort is correct

We can show the correctness by induction.

The base case is trivial for a list with its length being 1, i.e., \( p=r \).

Assume that it sorts lists correctly when the length is less than \( n \), and now considers a list with its length being \( n \).

We notice that after the partition in line 2, the size of both \( A[p, q-1] \) and \( A[q+1, r] \) are strictly less than \( n \). (?:) Thus, by the inductive assumption, the two sorts in lines 3 and 4 will sort \( A[p, q-1] \) and \( A[q+1, r] \).

Finally, since \( A[q] \) is in the “right” place after line 2, we are done. 😊

**Homework:** Exercises 7.1-3, and 7.2-2.
How much time does it take?

The running time of the quickSort depends on whether the partition leads to a “balanced” cut, which depends on the choice of the pivot.

If the *mean* element of a list is always chosen as the pivot, the partitioning will produce two balanced subproblems, each of which contains *about* \( n/2 \) items, then, quicksort is fast, since the associated recurrence will be

\[
T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).
\]

This is the same as what we came up with the merge sort (Cf. Page 53 of Chapter 2 notes).

\[
T(n) = O(n \log n). \smiley
\]

**Homework:** Think about 7.2-3.
The worst case

On the other hand, if either the maximum or the minimum is always chosen as the pivot, the partitioning produces one subarray with \( n - 1 \) elements and another one with 0 element.

If this occurs all the time, we have the following recurrence, since the partitioning itself costs \( \Theta(n) \):

\[
T(n) = T(n - 1) + T(0) + \Theta(n) \\
= T(n - 1) + \Theta(n) \overset{?}{=} \Theta(n^2).
\]

We end up with the same quadratic running time as that of the insertion sort.

This occurs when the list is either completely sorted, reversely sorted (Exercise 7.2-3), or everything is the same using \textit{this} \ PARTITION as discussed on Page 4, thus the need for Exercise 7.1-2.
The average case

Fortunately, the average-case running time of the quickSort is much closer to that for its best case, i.e., \( \Theta(n \log n) \). 😊

Recall that, if a partitioning always cuts off a constant share of the list, like what we did for the Binary Search, then the depth of the partitioning, i.e., the maximum number of partition that we can make, will be \( O(\log n) \). (Cf. Page 49 of Order of Growth notes)

Thus, the total time of quickSort is \( O(n \log n) \), as each partitioning takes \( O(n) \), by making \( n-1 \) comparisons.

For example, even if the partitioning always produces a 9-to-1 proportional split, we will get the following recurrence equation:

\[
T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn.
\]
What does it look like?

Below shows the situation.

With the leftmost branch, we have

\[ \frac{n}{10^k} \geq 1, \text{ i.e., } 10^k \leq n, \]

which gives,

\[ k \leq \log_{10} n. \]

when \( n = 100 \), we have \( k = 2 \); and when \( n = 1,000,000 \), \( k = 6 \).
What is going on?

Every level of the partitioning costs precisely $cn$, until the branch corresponding to the $\frac{1}{10}$ ratio can no longer be further cut at level $\log_{10} n$ ($= \Theta(\log n)$). From this point on, the partitioning will cost less than $cn$, since there are less than $n$ elements left for partitioning.

The cut-off process will continue until the longest branch corresponding to the $\frac{9}{10}$ ratio also cannot be further cut at level $\log_{\frac{9}{10}} n$ ($= \Theta(\log n)$).

More specifically, the size of this longest branch after each cut is the following: $\frac{9}{10} n$, $(\frac{9}{10})^2 n$, $\ldots$, $(\frac{9}{10})^k n$, $\ldots$, 1. We must have

$$
\left(\frac{9}{10}\right)^k n \geq 1,
$$

which leads to the result that

$$
k \leq \log_{\frac{9}{10}} n = \frac{\log_{10} n}{1 - \log_{10} 9} \approx 20 \log_{10} n = \Theta(\log n).
$$

Check out Page 13 of the *Heapsort* notes.
A little conclusion

We now know that this “unbalanced” partitioning can be partitioned at most \( \log n \) times, each of which is involved with at most \( cn \) comparisons. Hence, the total cost of the quickSort, even in this unbalanced partitioning, is also \( O(n \log n) \).

*In general, whenever a constant proportion is cut at all the partitioning, the quickSort always runs at \( O(n \log n) \).*

**Question:** How do we find our luck?

**Answer:** Go random. 😊

**Question:** Remember what I asked you to do when going through this sorting algorithm the first time?

**Answer:** Pick up a number without thinking… thus the result would be a random one.
A randomized version

When analyzing the average case of performance for the *InsertionSort* algorithm, we assumed that all permutations of the input numbers are equally likely. This is usually not true in practice. See the link on probabilistic distribution on the course page.

What we can do, though, is to add a randomization piece to our algorithm to obtain a good average-case performance on all the possible inputs, just as what we did in the probabilistic analysis chapter, as well as in Projects 3.

Such a randomized version of the quickSort algorithm, with a little patch, has become *the sorting algorithm of choice*, for sorting large number of elements. For example, quickSort was behind the `Arrays.sort()` method in *Java*, before it is replaced with *TimSort*.

Check out the course page for various references.
What should be the pivot?

In the earlier presentation, we always use the last element as the pivot, which is definitely not ideal. It could be either the smallest or the biggest one, thus leading to the worst partition. 😞

What we could do is to apply the technique of random sampling: instead of using $A[r]$, we randomly choose an element, switch it with $A[r]$, and then apply the original algorithm.

For example, once `java.util.Random` is imported, to randomly select an index in between 1 and 100, we simply do the following:

```java
rIndex=randomGenerator.nextInt(100)+1;
```

We now can expect the split of the list will be a “balanced” one, more about this later in Chapter 12.
The code

RANDOMIZED-PARTITION(A, p, r)
1. i<-RANDOM(p, r)
2. exchange(A[i], A[r])
3. return PARTITION(A, p, r)

Now, we present the new quickSort algorithm.

RANDOMIZED-QUICKSORT(A, p, r)
1. if p<r
2. then q<-RANDOMIZED-PARTITION(A, p, r)
3. RANDOMIZED-QUICKSORT(A, p, q-1)
4. RANDOMIZED-QUICKSORT(A, q+1, r)

An issue: The list will be almost sorted near the end, thus falling into the worst case.

Question: What should be the tissue?

Answer: Wrap up the smaller ones with Insertion sort, which works best when the list is (almost) sorted.
The average case

Assuming all sizes of $S_1$, the first part, are equally likely. Let $T(n, i)$ be the average time for quick sorting a list of size $n$ with $|S_1| = i$.

$$
\overline{T}(n) = \frac{1}{n} \sum_{i=0}^{n-1} [T(n, i) + cn]
= \frac{1}{n} \sum_{i=0}^{n-1} [\overline{T}(i) + \overline{T}(n-1-i)] + cn
= \frac{2}{n} \sum_{i=0}^{n-1} \overline{T}(i) + cn.
$$

By applying the telescoping technique (Cf. Pages 11-13, BST chapter for the derivation process.),

$$
\frac{\overline{T}(n)}{n+1} = \frac{1}{2} + 2 \sum_{i=3}^{n+1} \frac{1}{i} = 2H_n + O(1)
= 2 \ln n + O(1). \quad \text{(Cf. Page 5, Math Review)}
$$

$\overline{T}(n)$, the average-case running time of quick-Sort, is thus $\Theta(n \log(n))$.  

21