Chapter 7
Quicksort

Remember that Quicksort is also based on the divide-and-conquer strategy?

To quicksort a subarray $A[p..r]$, when it contains more than one element, 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 (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that everything in the first subarray is less then or equal to the pivot; and everything in the second is larger.

3) Recursively sort the two subarrays.

We are done, as no combination is needed.
The code

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

\[
\text{QUICKSORT}(A, p, r)
\]
1. //If more than one element
2. if \(p<r\)
3. //q gives where the pivot is
4. then \(q<-\text{PARTITION}(A, p, r)\)
5. \hspace{1em} \text{QUICKSORT}(A, p, q-1)
6. \hspace{1em} \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 \(\text{PARTITION}\) is the only piece that we need to go through.
The partition procedure

This procedure is at the heart of the whole algorithm, which rearranges the subarray $A[p..r]$ around a pivot, $A[q]$.

Below is one implementation, and you will have to do another one as an exercise (7.1-2.) that will do better when everything is the same.

PARTITION($A, p, r$)
1. $x \leftarrow A[r]$
2. $i \leftarrow p-1$
3. for $j \leftarrow p$ to $r-1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i+1$
7. exchange($A[i+1], A[r]$)
8. return $i+1$

**Question:** How does it do it?
An example

Below shows the action of this partition procedure.

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 segments: \( A[p..i] \) 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 \( A[r] \) contains the pivot.

Hence, the loop invariant for the for loop in lines 3–6 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.

At the very beginning, \( i = p - 1 \) and \( j = p \). Hence, we have that \( A[p, i] = A[p, p-1] = \emptyset \); \( A[i+1, j-1] = A[p, p-1] = \emptyset \); \( A[j, r-1] = A[p, r-1] \), i.e., nothing is known.

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 $A[j_0] \leq x$.

- $A[j_0] > x$: 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 $j_0 + 1$.

- $A[j_0] \leq x$: 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, the third segment becomes empty, namely, everything except the pivot has been categorized into either the first or the second segment.

We finally swap, in Lines 7 and 8, 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 that is not less than the pivot.

The important thing is that the pivot is in the right place.

**Homework:** Exercises 7.1-3(\*) and 7.1-4.
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.
Performance issues

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 \textit{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, and we have that

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

\textbf{Homework:} Exercises 7.2-2(*) and 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 \( \Theta(n) \):

\[
T(n) = T(n - 1) + T(0) + \Theta(n) \\
= T(n - 1) + \Theta(n) = \Theta(n^2)
\]

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

This indeed will occur when the list is either completely sorted, reversely sorted, or everything is the same using this PARTITION, thus the need for 7.1-2.
The average case

Fortunately, the average-case running time of the quicksort is much closer to that for its best case.

Recall that if a partitioning always cuts off a constant share of the list, then the depth of the partitioning, i.e., the maximum cuts, will be $O(\log n)$, which leads to an $O(n \log n)$ running time, as each partitioning takes $O(n)$.

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.
What is going on?

Every level of the partitioning costs precisely $cn$, until the branch corresponding to the $\frac{1}{10}$ 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. This will continue until the longest branch corresponding to the $\frac{9}{10}$ 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: $n\frac{9}{10}, n\left(\frac{9}{10}\right)^2, \ldots, n\left(\frac{9}{10}\right)^k, \ldots, 1$. We must have

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

which leads to the result that

$$k \leq \log_{\frac{9}{10}} n = \Theta(\log n).$$
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.
A randomized version

When analyzing the average case of performance for the Insertion sort algorithm, we assumed that all permutations of the input numbers are equally likely. This is usually not true in practice.

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 last project.

Such a randomized version of the quicksort algorithm has become the sorting algorithm of choice for large number of elements.
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.

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.
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)
The average case

We assume that 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 $|S_1| = i$, we have the following:

$$
\overline{T}(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(n, i)
$$

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

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

We have that, by applying the telescoping technique,

$$
\frac{\overline{T}(n)}{n + 1} = \frac{1}{2} + 2 \sum_{i=3}^{n+1} \frac{1}{i} = 2\mathcal{H}_n + O(1)
$$

$$
= 2 \ln n + O(1).
$$

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

19