Quicksort

(CLRS 7)

- We saw the divide-and-conquer technique at work resulting in Mergesort:
  - Partition n elements array A into two subarrays of $n/2$ elements each
  - Sort the two subarrays recursively
  - Merge the two subarrays

  Running time: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$. Mergesort is not in-place.

- Note that dividing the array is easy (basically there's nothing to do beyond computing the middle index), and the work for mergesort happens in the "merge" phase.

- Why not do it the other way around? Divide the elements such that there is no need of merging, that is
  - Partition $A[1...n]$ into subarrays $A' = A[1..q]$ and $A" = A[q+1...n]$ **such that all elements in $A"$ are larger than all elements in A'.**
  - Recursively sort A' and $A"$.

  Here the work happens in the "divide" phase, and there is no need to to combine/merge because of the way A was partitioned, A is already sorted after sorting A' and $A"$.

- Let's ignore for the moment how partition would actually work —-we'll do that in a second. Assuming someone gives us a Partition function that does what's expected, then the pseudo code for QUICKSORT would look like this. As usual with recursive functions on arrays, we see indices p and r as arguments.

```
QUICKSORT(A, p, r)
IF p < r THEN
    q=PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

  To sort the entire array, one would need to call QUICKSORT$(A, 0, n-1)$.

- Analysis: What can we expect in terms of running time? If $q = n/2$ and we partition in $\Theta(n)$ time, we again get the recurrence $T(n) = 2T(n/2) + \Theta(n)$ for the running time $\Rightarrow T(n) = \Theta(n \log n)$

To get Quicksort to run in $O(n \log n)$ running time, we'll have to develop a partition algorithm which divides A in (approximately) two halves. We'll come back to this.

- Correctness: Let's assume that PARTITION works correctly (which can be shown separately, once we actualul discuss how to do it). Then:

  1. We start by showing that Quicksort sorts correctly when $n = 1$ that is when there's only one element in the array. [Show]
  2. Now let's show that Quicksort sorts correctly when $n = 2$: Partition will partition into A[1] and A[2], which are both arrays of size 1, which are sorted correctly because of (1).
  3. Now if $n = 3$: Partition will partition into an array of size 1 and one of size 2, or the other way around. These are sorted correctly because of (1) and (2).
  4. And so on.

  This is the informal, bottom-up argument, but it captures the spirit. The formal way to prove this by induction. Basically, if Partition works correctly and Quicksort sorts correctly the two sides of the partition (this is the inductive hypothesis), then it follows that the whole array is sorted correctly. If you know induction, this should be easy.

Partition

Now that we got the big picture, let's dive into how to partition.

- First, note that it would be pretty eay to partition if we did not have to do it in place (think, think..).
- The trick is doing it in-place. If we are able to come up with an in-place partition, this immediately leads to in-place quicksort. In the sorting world, in-placeness is a nice property to have..
- Below is the code from the textbook (CLRS). It is called LOMUTO partition, named after the researcher who first described it.

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
FOR j = p TO r − 1 DO
    IF A[j] ≤ x THEN
        i = i + 1
        Exchange A[i] and A[j]
    FI
OD
Exchange A[i + 1] and A[r]
RETURN i + 1
```

- Example:

|	2	8	7	1	3	5	6	4	i=0, j=1
2	|	8	7	1	3	5	6	4	i=1, j=2
2	|	8	|	7	1	3	5	6	4	i=1, j=3
2	|	8	7	|	1	3	5	6	4	i=1, j=4
2	1	|	7	8	|	3	5	6	4	i=2, j=5
2	1	3	|	8	7	|	5	6	4	i=3, j=6
2	1	3	|	8	7	5	|	6	4	i=3, j=7
2	1	3	|	8	7	5	6	|	4	i=3, j=8
2	1	3	|	4	|	7	5	6	8	q=4

- Correctness: PARTITION can be proved correct using the loop invariants:
  - $A[k] \leq x$ for $p \leq k \leq i$
  - $A[k] > x$ for $i+1 \leq k \leq j-1$
  - $A[k] = x$ for $k = r$

  The invariants are trivially true before the first iteration of the loop (when $i = p-1, j = p$).

  Consider the jth iteration of the loop. Assume that the invariants are true before the execution of the loop. We need to argue that $A[j]$ is inserted in the "right" place, which means that the invariants are maintained for the next iteration, when j is incremented. [Argue]

  Thus the invariants start as being true, then every iteration of the loop maintains them, so they will be true at the end after the last iteration, when $j = r-1$. Then $A[i+1]$ is swapped with $A[r]$. This means partition works correctly.

- PARTITION analysis: PARTITION runs in time $\Theta(r-p)$ (does one pass through the input), which is linear in the number of elements that are partitioned.

QUICKSORT analysis

- Running time depends on how well PARTITION divides A.
- In the example above it does reasonably well.
- If array is always partitioned nicely in two halves (partition returns $q = \frac{r+p}{2}$), we have the recurrence $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \lg n)$.
- But, in the worst case PARTITION always returns $q = p$ or $q = r$ and the running time becomes $T(n) = \Theta(n) + T(0) + T(n-1) \Rightarrow T(n) = \Theta(n^2)$.
- And maybe even worse, the worst case is when A is already sorted.
- Let's dig a step further. So all-good-splits give us $\Theta(n \lg n)$ and all-bad-splits give us $\Theta(n^2)$. What about a combination of good and bad splits?

- Note that a good split is not necessarily a perfectly balanced one ($\frac{1}{2}$-to-$\frac{1}{2}$). Any split that puts a constact fraction of the elements to one side is good.
  - Example: Split is $\frac{9}{10}n$, $\frac{1}{10}n$.
    $T(n) = T(\frac{9}{10}n) + T(\frac{1}{10}n) + n$
    Solution: $\Theta(n \lg n)$
- So splits like $\frac{1}{2}$-to-$\frac{1}{2}$, $\frac{1}{3}$-to-$\frac{2}{3}$, ,,, $\frac{1}{9}$-to-$\frac{8}{9}$,.... are all good. There are a LOT of good splits!!!
- Even if every only other split is good, we can show that Quicksort still performs well (a bad split is absorbed into a good split).
- Intuitively, there are A LOT of cases where quicksort will perform in $\Theta(n \lg n)$ time. Can we theoretically justify this?

Average running time

Formally, we want to find what is the average case running time of QUICKSORT. Is it close to worst-case ($\Theta(n^2)$), or to the best case $\Theta(n \lg n)$? We have to be careful here with the set of inputs on which we take the average, because average time depends on the distribution of inputs for which we take the average.

- If we run QUICKSORT on a set of inputs that are all almost sorted, the average running time will be close to the worst-case.
- Similarly, if we run QUICKSORT on a set of inputs that give good splits, the average running time will be close to the best-case.
- Most of the time, people analyse average time assuming a uniform distribution of the input.

What happens if we run QUICKSORT on a set of inputs which are picked uniformly at random from the space of all possible input permutations?

- Then the average case will also be close to the best-case. Why? Intuitively, if any input ordering is equally likely, then we expect at least as many good splits as bad splits, therefore on the average a bad split will be followed by a good split, and it gets "absorbed" in the good split.

So, under the assumption that **all input permutations are equally likely**, the average time of QUICKSORT is $\Theta(n \lg n)$.

- This can be proved formally, but we won't do it in this class.

Is is realistic to assume that all input permutations are equally likely?

- Not really. In many cases the input is almost sorted (e.g. rebuilding index in a database etc).

If the assumption of uniform distribution is not realistic, then the fact that Quicksort has a good average time does not help us that much. How can we make QUICKSORT have a good average time irrespective of the input distribution?

YES, using **randomization**!! See below.

Randomization

We consider what are called *randomized algorithms*, that is, algorithms that make some random choices during their execution.

- Running time of a normal *deterministic* algorithm only depends on the input.
- Running time of a randomized algorithm depends not only on input but also on the random choices made by the algorithm.
- Running time of a randomized algorithm is not fixed for a given input!
- Randomized algorithms have best-case and worst-case running times, but the inputs for which these are achieved are not known, they can be any of the inputs.

We are normally interested in analyzing the *expected* running time $T_e(n)$ of a randomized algorithm, that is, the expected (average) running time for all inputs of size n. Here $T(X)$ denotes the running time on input X (of size n)

$$T_e(n) = E_{|X|=n}[T(X)]$$

Randomized Quicksort

There are two ways to go about randomizing Quicksort.

1. The first one is: We can enforce that all $n!$ permutations are equally likely by randomly permuting the input before the algorithm.
   - Most computers have pseudo-random number generator $random(1, n)$ returning "random" number between 1 and n
   - Using pseudo-random number generator we can generate a random permutation (such that all $n!$ permutations equally likely) in $O(n)$ time:
     Choose element in $A[1]$ randomly among elements in $A[1..n]$, choose element in $A[2]$ randomly among elements in $A[2..n]$, choose element in $A[3]$ randomly among elements in $A[3..n]$, and so on.
2. Alternatively, we can modify PARTITION slightly and exchange last element in A with random element in A before partitioning.

```
RANDPARTITION(A, p, r)
i=RANDOM(p, r)
Exchange A[r] and A[i]
RETURN PARTITION(A, p, r)
```

```
RANDQUICKSORT(A, p, r)
IF p < r THEN
    q=RANDPARTITION(A, p, r)
    RANDQUICKSORT(A, p, q − 1)
    RANDQUICKSORT(A, q + 1, r)
```

It can be shown formally that the expected running time of randomized quicksort (on inputs of size n) is $\Theta(n \lg n)$.

- We do not do it in this class, but I encourage you to read the text book!

NOTES

- In a future lecture (selection) we will see how to make quicksort run in worst-case $O(n \log n)$ time.
- Trivia: Quicksort was invented by Sir Charles Antony Richard Tony Hoare, a British computer scientist and winner of the 1980 Turing Award, in 1960, while he was a visiting student at Moscow State University. While in Moscow he was in the school of Kolmogorov (well known mathematician).
- The original Quickort was described with a different partition algorithm, one that works from both ends; this is referd to as Hoare-partition (after Hoare who invented Quicksort). We'll look at this different way to partition in the lab.
- Both Hoare's and Lomuto partitions are in-place, but none of them is stable.
- Thus, in an efficient implementation, with an in-place partition, Quicksort is NOT stable.
- I don't know of any practical comparison of the two partitions, and I would guess they are both equally fast in practice. Lomuto-partition may be considered simpler and better, because it uses a single pointer (i, to point to the last element left of pivot)—and this is why textbooks prefer it. OTOH Hoare-partition does well (ie $O(n \lg n)$) when all elements are equal, while Lomuto-partition is quadratic.
- The idea behind Lomuto's partition can be extended to deal with elements equal to the pivot (place them in the middle instead of carrying them through recursion); also with 3-way partitioning.
- Quicksort can be made stable by using extra space (thus not in place anymore).