Quicksort

(CLRS 7)

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

  Running time: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$
- Another possibility is to 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 $A"$.
  - (no need to to combine/merge. A already sorted after sorting A' and $A"$)
- Pseudo code for Quicksort:

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

  Sort using Quicksort$(A, 0, n-1)$

  If $q = n/2$ and we divide in $\Theta(n)$ time, we again get the recurrence $T(n) = 2T(n/2) + \Theta(n)$ for the running time $\Rightarrow T(n) = \Theta(n \log n)$

  The problem is: can we develop a partition algorithm which always divide A in two halves?

  Quicksort **correctness**:

  - Assume that Partition works correctly (that can be shown separately). Quicksort correctness can be shown, inductively.

Partition

```
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
```

- PARTITION can be proved correct (by induction) 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 $k = r$

  These are true before the execution of the loop, when $i = p-1, j = p$; and are true after every execution of the loop. Thus at the end when $j = p-1$ this means partition works correctly.

- Analysis: PARTITION runs in time $\Theta(r-p)$ (does one pass through the input)

QUICKSORT analysis

- Running time depends on how well PARTITION divides A.
- In the example 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 what is maybe even worse, the worst case is when A is already sorted.
- So why is it called "quick"-sort? Because it "often" performs very well—can we theoretically justify this?
- Even if all the splits are relatively bad, we get $\Theta(n \log n)$ time:
- 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: $\Theta(n \lg n)$
- Even if every otehr split is balanced, 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

The natural question is: what is the average case running time of QUICKSORT? Is it close to worst-case ($\Theta(n^2)$, or to the best case $\Theta(n \lg n)$? 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.
- 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 here.

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).

The question is: how can we make QUICKSORT have a good average time irrespective of the input distribution? Using **randomization**.

Randomization

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

- Running time of normal *deterministic* algorithm only depend 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 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

- 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.
- 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 that the expected running time of randomized quicksort (on inputs of size n) is $\Theta(n \lg n)$.

Next time we will see how to make quicksort run in worst-case $O(n \log n)$ time.