Quicksort

(CLRS 7)

- We previously saw how the divide-and-conquer technique can be used to design sorting algorithm—Merge-sort
  - 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”.
  - (nothing 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($A, q + 1, r$)
FI
```

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

  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 that it is hard to develop partition algorithm which always divide A in two halves

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

QUICKSORT **correctness**:

- ..easy to show, inductively, if PARTITION works correctly
- 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 invariant:
  - $A[k] \leq x$ for $p \leq k \leq i$
  - $A[k] > x$ for $i + 1 \leq k \leq j - 1$
  - $A[k] = x$ for $k = r$

QUICKSORT **analysis**

- PARTITION runs in time $\Theta(r - p)$
- 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?
      Guess: $T(n) \leq cn \log n$
      Induction

$$\begin{aligned} T(n) &= T(\frac{9}{10}n) + T(\frac{1}{10}n) + n \\ &\leq \frac{9cn}{10}\log(\frac{9n}{10}) + \frac{cn}{10}\log(\frac{n}{10}) + n \\ &\leq \frac{9cn}{10}\log n + \frac{9cn}{10}\log(\frac{9}{10}) + \frac{cn}{10}\log n + \frac{cn}{10}\log(\frac{1}{10}) + n \\ &\leq cn\log n + \frac{9cn}{10}\log 9 - \frac{9cn}{10}\log 10 - \frac{cn}{10}\log 10 + n \\ &\leq cn\log n - n(c\log 10 - \frac{9c}{10}\log 9 - 1) \end{aligned}$$

      $T(n) \leq cn\log n$ if $c\log 10 - \frac{9c}{10}\log 9 - 1 > 0$ which is definitely true if $c > \frac{10}{\log 10}$
  - So, in other words, if the splits happen at a constant fraction of n we get $\Theta(n \lg n)$—or, it's almost never bad!

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 $\Theta(n \lg n)$ (intuitively). Is this assumption realistic?

- Not really. In many cases the input is almost sorted; think of rebuilding indexes 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

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

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.

    Note: Just choosing $A[i]$ randomly among elements $A[1..n]$ for all i will not give random permutation! Why?

- 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($A, q + 1, r$)
FI
```

Expected Running Time of Randomized Quicksort

Let $T(n)$ be the running time of RandQuicksort for an input of size n.

- Running time of RandQuicksort is the total running time spent in all Partition calls.
- Partition is called n times
    - The pivot element x is not included in any recursive calls.
- One call of Partition takes $O(1)$ time plus time proportional to the number of iterations of FOR-loop.
    - In each iteration of FOR-loop we compare an element with the pivot element.

  ⇓

  If X is the number of comparisons $A[j] \leq x$ performed in Partition over the entire execution of RandQuicksort then the running time is $O(n+X)$.

  ⇓

  $E[T(n)] = E[O(n+X)] = n + E[X]$

  ⇓

  To analyze the expected running time we need to compute $E[X]$

- To compute X we use $z_1, z_2, \ldots, z_n$ to denote the elements in A where z_i is the ith smallest element. We also use Z_{ij} to denote $\{z_i, z_{i+1}, \ldots, z_j\}$.
- Each pair of elements z_i and z_j are compared at most once (when either of them is the pivot)

  ⇓

  $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ where

  $$X_{ij} = \begin{cases} 1 & \text{If } z_i \text{ compared to } z_i \\ 0 & \text{If } z_i \text{ not compared to } \mathrm{z_i} \end{cases}$$

  ⇓

  $$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ compared to } z_j] \end{aligned}$$

- To compute $Pr[z_i \text{ compared to } z_j]$ it is useful to consider when two elements are *not* compared.

  > Example: Consider an input consisting of numbers 1 through n.
  >
  > Assume first pivot it 7 ⇒ first partition separates the numbers into sets $\{1,2,3,4,5,6\}$ and $\{8,9,10\}$.
  >
  > In partitioning, 7 is compared to all numbers. No number from the first set will ever be compared to a number from the second set.

  In general, once a pivot x, $z_i < x < z_j$, is chosen, we know that z_i and z_j cannot later be compared.

  On the other hand, if z_i is chosen as pivot before any other element in Z_{ij} then it is compared to each element in Z_{ij}. Similar for z_j.

In example: 7 and 9 are compared because 7 is first item from $Z_{7,9}$ to be chosen as pivot, and 2 and 9 are not compared because the first pivot in $Z_{2,9}$ is 7.

Prior to an element in Z_{ij} being chosen as pivot, the set Z_{ij} is together in the same partition $\Rightarrow$ any element in Z_{ij} is equally likely to be first element chosen as pivot $\Rightarrow$ the probability that z_i or z_j is chosen first in Z_{ij} is $\frac{1}{j-i+1}$

$\Downarrow$

$Pr[z_i \text{ compared to } z_j] = \frac{2}{j-i+1}$

- We now have:

$$\begin{array}{rcl} E[X] & = & \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr[z_i \text{ compared to } z_j] \\ & = & \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ & = & \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\ & < & \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k} \\ & = & \sum_{i=1}^{n-1} O(\log n) \\ & = & O(n\log n) \end{array}$$

- Since best case is $\theta(n\lg n) \Longrightarrow E[X] = \Theta(n\lg n)$ and therefore $E[T(n)] = \Theta(n\lg n)$.

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