CSOR W4231.002 **Lecture 2: Feb 10, 2014** **Spring 2014**

Today we discuss another **divide and conquer** algorithm for sorting, namely Quicksort. Quicksort is the standard algorithm used for sorting and is an **in-place** algorithm. Although its worst-case running time is $\Theta(n^2)$, its average-case running time is $\Theta(n \log n)$.

We will use Quicksort to introduce randomized algorithms and discuss a randomized variant of this algorithm that has expected running time $\Theta(n \log n)$.

1 Quicksort

The main idea of Quicksort is as follows: at every recursive call, pick an element from the input. We call this element the **pivot** element. We will place *pivot* in its final location in the sorted array as follows: we re-organize the array so that all elements smaller than *pivot* are to the left of *pivot* and all items larger than *pivot* are to its right (see Figure 1(a)). Then recursively sort the left subarray of A, where all items are smaller than *pivot*, and the right subarray of A, where all items are greater than *pivot*. The pseudocode follows.

Quicksort $(A, left, right)$ (originally call Quicksort$(A, 1, n)$)
  if $|A| = 0$ **then** return $//A$ is empty
  end if
  $split = \text{Partition}(A, left, right)$
  Quicksort $(A, left, split - 1)$
  Quicksort $(A, split + 1, right)$

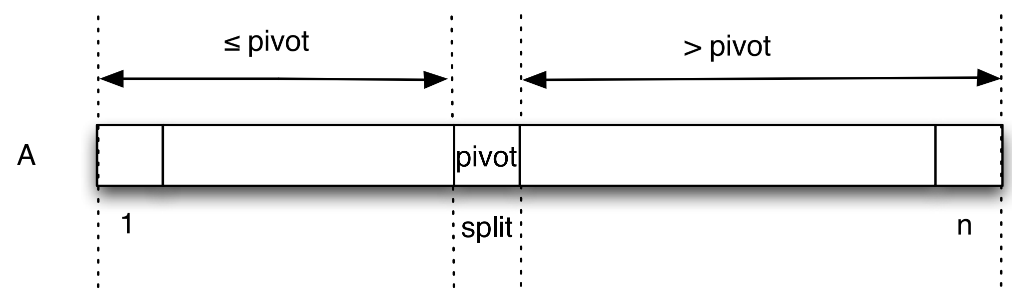

Figure 1: Re-organized array A when Partition$(1, n)$ returns.

1.1 Subroutine Partition

Subroutine Partition picks a *pivot* element and re-organizes $A(left, right)$ so that all elements before the *pivot* element are smaller than it while all elements after the *pivot* element are greater than it. The subroutine returns the position of *pivot* in the re-organized array (*split*).

First we have to decide how to pick the pivot element that will be used to split the input into two parts. We assume that we always pick the last element of the input to Partition [1] as the pivot element, i.e., $pivot = A[right]$. Hence $A[right]$ will be placed at its final location in the sorted array when the subroutine returns.

[1] There is nothing particular about $A[right]$: alternatively, we could pick any fixed $A[j]$ for $left \leq j \leq right$ as our pivot element .

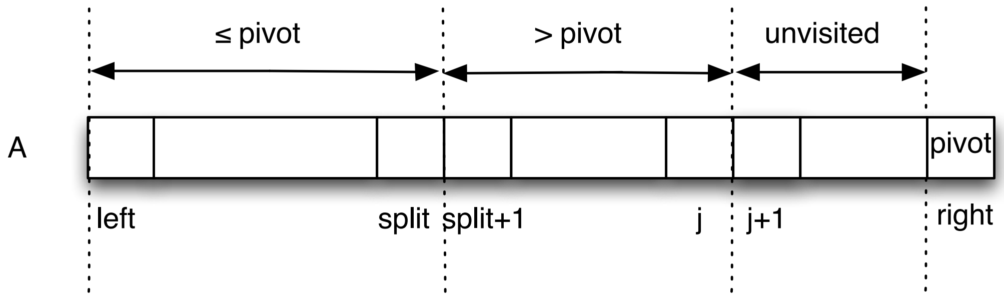

Figure 2: The three regions maintained by subroutine Partition.

In order to implement Partition **in place** some care is required.

Intuitively, Partition examines the elements in its input one by one and maintains three regions in $A(left, right)$ (see Figure 2(a)).

Specifically, after examining the j-th element for $left \leq j \leq right - 1$, these regions are as follows:

1. The first region, located at the left of $A[left, right]$, indicates all elements encountered so far that are smaller than *pivot*. We maintain a pointer *split* that points to the last element of this region. Hence this region starts at $A[left]$ and ends at $A[split]$.

2. The second region, to the right of *split*, indicates all elements encountered so far that are greater than *pivot*. This region starts at $A[split + 1]$ and ends at $A[j]$.

3. The third region contains the unvisited elements, that is, the elements we have not encountered so far. It starts at element $A[j + 1]$ and ends at element $A[right - 1]$.

At every iteration we compare the first element from the third region with *pivot*.
If the element is smaller, then we swap this element with element $A[split + 1]$: this is the first element of the second region, hence is greater than *pivot*; and we increment *split* to account for the new element in the first region. So now all elements smaller than *pivot* are grouped together. The pseudocode for Partition follows.

Partition$(A, left, right)$
  $pivot = A[right]$
  $split = left - 1$
  for $j = left$ to $right - 1$ **do**
    if $A[j] \leq pivot$ **then**
      swap$(A[j], A[split + 1])$
      $split = split + 1$
    end if
  end for
  swap$(pivot, A[split + 1])$ //Place *pivot* right after $A[split]$
  return $split + 1$ // the element at $split + 1$ is at its final location in the sorted array

1.1.1 Analysis of Partition: Correctness

We will prove correctness by induction on j. The statement we want to prove is the following.

Claim 1 *For all $left \leq j \leq right - 1$, at the end of loop j, all elements from $A[left, \ldots, j]$ that are less or equal to pivot are located in positions $A[left, \ldots, split]$; and all elements from $A[left, \ldots, j]$ that are greater than pivot are located in positions $A[split + 1, \ldots, j]$.*

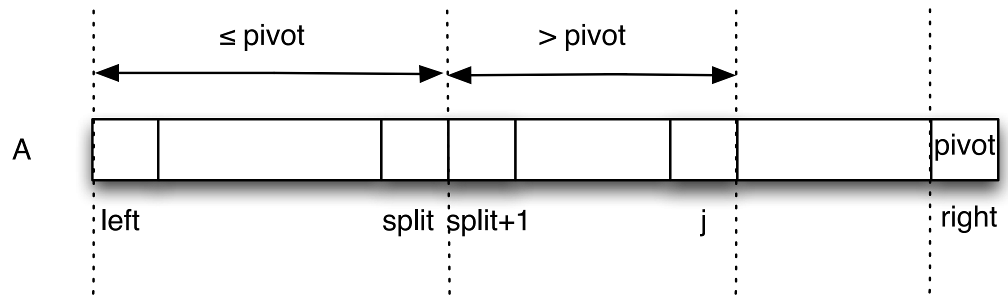

Figure 3: Induction hypothesis for Claim 1.

Note that if the claim is true, correctness follows: by the claim, all elements from $A[left, \ldots, right-1]$ that are less than *pivot* are located in positions $A[left, \ldots, split]$; and all elements from $A[left, \ldots, right-1]$ that are greater than *pivot* are located in positions $A[split+1, \ldots, right-1]$. After the for loop, elements $A[split+1]$ and *pivot* are swapped: thus the last element that is less than *pivot* is still at position *split* while the first element that is greater than *pivot* is now at position $split+2$, while *pivot* occupies position $split+1$.

Proof. We will show the claim by induction on j, for $left \leq j \leq right-1$.

- **Base case:** when $j = left$, during the first execution of the for loop, if $A[left] \leq pivot$ then $A[left]$ is swapped with itself and *split* becomes $left$. Otherwise nothing happens. In either case, the claim holds.
- **Induction hypothesis:** Assume that the claim is true for some $left \leq j < right-1$ (see Figure 3(a)).
- **Induction step:** We will show it true for $j+1$. At the beginning of loop $j+1$, by the induction hypothesis, $A[left, \ldots, split]$ are all less or equal to *pivot* and $A[split+1, \ldots, j]$ are all greater than *pivot*. During the $j+1$ loop there are two possibilities:
  1. $A[j+1] \leq pivot$: Now $A[j+1]$ is swapped with $A[split+1]$; at this point, $A[left, \ldots, split+1]$ are all less or equal to *pivot* and $A[split+2, \ldots, j+1]$ are all greater than *pivot*. Incrementing *split* (the next step in the conditional statement) yields that the claim holds at the end of loop $j+1$.
  2. $A[j+1] > pivot$: nothing is done. The truth follows from the induction hypothesis.

In both cases, the claim holds. □

1.1.2 Partition: running time

Partition requires $\Theta(n)$ time: go through each of the $n-1$ elements once and perform constant amount of work for each element.

1.2 Analysis of Quicksort: correctness

By strong induction on the size of the array $n \geq 0$.

- **Base case:** for $n = 0$, Quicksort does nothing (an empty subarray needs not be sorted).
- **Induction hypothesis:** Assume that Quicksort correctly sorts for all $0 \leq m < n$.

- **Induction step:** We will show that Quicksort correctly sorts an array of size n.
  Since Partition$(A, 1, n)$ is correct, it will return an index *split* and a re-organized array A such that all elements in $A[1, \ldots, split - 1]$ are less or equal to $A[split]$ and all elements in $A[split + 1, \ldots, n]$ are greater than $A[split]$.

  By the induction hypothesis, Quicksort$(A, 1, split - 1)$ and Quicksort$(A, split + 1, n)$ each returns a permutation of its input subarray that is correctly sorted since $split - 1, n - split < n$. Therefore, $A[1] \leq \ldots \leq A[split - 1]$ and $A[split + 1] \leq \ldots \leq A[n]$. It follows that the whole array is sorted since $A[1] \leq \ldots \leq A[split - 1] \leq A[split] < A[split + 1] \leq \ldots \leq A[n]$.

1.3 Running time of Quicksort: best, worst and average case

The running time of Quicksort depends on which elements are used for the partitioning and how they compare to the rest of the elements in the input. This decides the sizes of the inputs to the two recursive calls and therefore the recurrence for the running time.

- **Best case:** Suppose that every call to Partition picks as *pivot* the *median* of its input. In this case, Partition always splits its input into two lists of almost equal sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, hence at most $n/2$. Then the recurrence becomes:

$$T(n) = 2T(n/2) + \Theta(n) = O(n \log n).$$

- **Worst case:** On the other hand, assume that every time Partition is called, the pivot element is bigger (or smaller) than every other element in the array, i.e., the array is already sorted. In this case, Partition returns one list of size $n - 1$ and one list of size 0. Unlike before, this partitioning is very unbalanced, in the sense that the sizes of the lists are very different. Let $T(0) = d$ for constant $d > 0$. Then for constant $c > 0$, we can show (e.g., by the substitution method) that

$$T(n) = T(n - 1) + T(0) + cn = \Theta(n^2).$$

  This is especially bad since the worst-case input is the sorted input.

  The above bound is tight since we can upper bound the running time of Quicksort as follows: here are at most n calls to Partition (why?) and each call requires $O(n)$ time, hence the worst-case running time is $O(n^2)$.

- **Average case:** Our input consists of n numbers. What is an "average" input to sorting?

  This depends on the application: in some applications all inputs may be equally probable, i.e., any of the $n!$ permutations of the n numbers is equally probable to be the input. In other applications the input may be almost sorted.

  In your book there is intuition why average-case running time of Quicksort is $O(n \log n)$.

From now on, we will focus on how to use randomness to provide Quicksort with a random input so that it exhibits its average case performance regardless of the ordering of the numbers in its original input. We start with a discussion on how randomness affects the analysis of the running times of algorithms.

2 Two ways of viewing randomness in computation

1. Our algorithm is **deterministic** but the world behaves **randomly**. (This is the kind of algorithms we've encountered so far.)

    - Given the same input, different executions of the algorithm spend the same time to produce the same output.

- The input is **randomly** generated according to some underlying distribution.
- We are interested in analyzing the running time of the algorithm on an average input (**average case** analysis).

2. Our algorithm behaves **randomly** and is provided the **worst-case** input.

   - Given the same input, different executions of the algorithm produce the same output but may spend different times to do so because the running time now depends on the **random choices** of the algorithm; to this end, the algorithm may flip coins or generate random numbers. We assume that our random samples are independent of each other.
   - The world provides its **worst-case** input.
   - We are interested in analyzing the running time of the randomized algorithm on such a worst-case input (**expected** running time).

Remark 1: Allowing the algorithm to make random choices strictly empowers the algorithm: e.g., a worst-case input may be transformed into a random input, i.e., one that is closer to an "average" input.
Remark 2: Deterministic algorithms are a special case of randomized algorithms.

3 Randomized Quicksort

How can we randomize Quicksort to make sure that it works with a random input even when it receives a worst-case input?

- **Answer 1**: We could explicitly permute the input: given the input, we generate a random permutation of it.
- **Answer 2**: Another way that yields a simpler analysis is to use *random sampling* for choosing the pivot element: instead of always choosing $pivot = A[right]$ we will randomly select an element from $A[left, right]$ as $pivot$.
  - **Intuition:** No matter how our input is organized, we won't often pick the largest or smallest element as our pivot element (unless we are really, really unlucky). Therefore we expect that most often the two subarrays that Partition returns will be "balanced" in size which should result in a running time close to the average case running time.

Below is our modified Partition procedure. Function random(a, b) returns a random number between a and b inclusive.

Randomized Partition$(A, left, right)$
  $b =$ random$(left, right)$
  swap$(A[b], A[right])$
  return Partition$(A, left, right)$

3.1 Discrete random variables

To analyze the expected running time of a randomized algorithm we need to keep track of certain parameters and their expected size over the random choices of the algorithm. To this end, we use random variables.
A discrete random variable takes on a finite number of values, each with some probability. Given a discrete random variable X, we will be interested in its expectation

$$E[X] = \sum_j j \cdot \Pr[X = j].$$

Example 1: Bernoulli trial Suppose we flip a biased coin that comes "heads" with probability p and "tails" with probability $1-p$. Let X be the random variable that has value 1 if the coin comes heads and 0 otherwise. Then $\Pr[X = 1] = p$ and $\Pr[X = 0] = 1-p$ and $E[X] = 1 \cdot \Pr[X = 1] + 0 \cdot \Pr[X = 0] = p$.

Indicator random variable: A discrete random variable that only takes on values 0 and 1. Basically such a random variable is used to denote the occurrence or non-occurrence of an event. In the example above, X is an indicator random variable and denotes the occurrence (or not) of "heads" in the coin flip. For indicator random variables, $E[X] = \Pr[X = 1]$.

Example 2: Bernoulli trials. Now suppose we flip the biased coin n times and we want to know what is the expected number of times we will get "heads".

In this case, we may define X to be the random variable counting the number of times that "heads" appears. We have

$$E[X] = \sum_{j=0}^{n} j \cdot \Pr[X = j].$$

We can deal with $\Pr[X = j]$ in a straightforward way: X follows the binomial distribution $B(n, p)$, that is $\Pr[X = j] = \binom{n}{j} p^j (1-p)^{n-j}$. Or, we can think about X as follows: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = 1 \text{ iff the outcome of the } i\text{-th coin flip is "heads".}$$

Define the random variable $X = \sum_{i=1}^{n} X_i$; we want $E[X]$. Note that X is a complicated random variable defined as the sum of simpler random variables, of which we know the expectation. In this case, we possess a powerful tool, namely linearity of expectation.

Proposition 1 *Let $X_1, \ldots, X_k$ be arbitrary random variables. Then*

$$E[X_1 + X_2 + \ldots + X_k] = E[X_1] + E[X_2] + \ldots + E[X_k]$$

Note that we do not need assume anything about the random variables. In particular, they do not need be independent. For example, if we want $E[X_1 + X_1^2]$, then clearly X_1, X_1^2 are dependent, but linearity of expectation applies, thus $E[X_1 + X_1^2] = E[X_1] + E[X_1^2]$.

Back to Example 2: we can now rewrite

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p = np.$$

3.2 Expected running time for Randomized QS

We will now analyze the expected running time $T(n)$ of Randomized Quicksort. This procedure differs from Quicksort only in how they select their pivot elements. Hence we will base our analysis of Randomized Quicksort on Quicksort and Partition.

We want to bound the expected running time $T(n)$. We begin with the following observations.

1. Partition can be called at most n times.

2. Let X be the total number of times that a comparison is performed at line 4 in all calls to Partition.

    - Each Partition spends constant amount of work outside the for loop and there are at most n calls to Partition; hence the total work spent outside the for loop in all calls to Partition is $O(n)$.

- There are X comparisons in total and each of them may require some further constant work (lines 5 and 6); hence the total amount of work spent inside the for loop in all calls to Partition is $O(X)$.

$\rightarrow$ Thus the running time of Randomized QS is $O(n + X)$.

3. Any two elements of the input will be compared at most once: comparisons are only performed with the pivot element of a call to Partition. This element is placed in its final location in the output during the call and not be part of the input to any future recursive call.

In order to bound $T(n)$ we need analyze X. To this end, we need to understand when two elements are compared. To simplify the analysis, instead of working with the input $x_1, \ldots, x_n$ we relabel it as $z_1, z_2, \ldots, z_n$, where z_i is the i-th smallest number. Assuming that all our input numbers are distinct, we have that $z_i < z_j$, for $i < j$.
Let X_{ij} be an indicator random variable such that

$$X_{i,j} = 1 \text{ iff items } z_i \text{ and } z_j \text{ are compared during QS.}$$

By observation 3, since there are n elements in the input, we have $\binom{n}{2} = \frac{n(n-1)}{2}$ distinct possible pairs to account for, thus equally many $X_{i,j}$'s. Thus

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}.$$

We want $E[X]$; by linearity of expectation,

$$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{i,j} = 1]$$

Hence we need compute $\Pr[X_{i,j} = 1]$. To this end, we need understand when z_i and z_j are compared, for $i < j$. We'll start by considering when they are **not** compared.

- If a Partition call picks as pivot an element z_k **outside** the set $Z_{i,j} = \{z_i, z_{i+1}, \ldots, z_j\}$, then
  - since neither z_i or z_j are the pivot elements, z_i and z_j are not compared;
  - furthermore, all elements in $Z_{i,j}$ will either be greater or smaller than the pivot element z_k and hence they will **all belong to the same subproblem after the split**.
- What happens when a call to Partition picks as pivot an element from the set $\{z_i, z_{i+1}, \ldots, z_j\}$ for the first time? There are 3 cases:
  1. Partition picks z_i as its pivot element: then z_i is compared with every element in $Z_{i,j} - \{z_i\}$, and in particular with z_j; after this call, z_i is placed in its final location in the sorted array and will not be encountered in any future calls to Partition.
  2. Partition picks z_j as its pivot element: then z_j is compared with every element in $Z_{i,j} - \{z_j\}$, and in particular with z_i; after this call, z_j is placed in its final location in the sorted array and will not be encountered in any future calls to Partition.
  3. Partition picks z_ℓ as its pivot element, for some $i < \ell < j$. Then z_i and z_j are never compared: they are not compared during this call to Partition and they will not be compared in the future since they will be placed into different subproblems.

Therefore, the only way for z_i and z_j to ever be compared is if they are the chosen as the pivot elements by the first Partition call that chooses its pivot element from $Z_{i,j}$. We are now ready to compute $\Pr[X_{i,j} = 1]$:

$\Pr[X_{i,j} = 1] = \Pr[z_i$ or z_j are chosen as *pivot* by the first Partition call that picks as *pivot* an element from $Z_{i,j}]$

Since these two events are mutually exclusive we obtain

$$\begin{aligned}
\Pr[X_{i,j} = 1] &= \Pr[z_i \text{ is chosen as } \textit{pivot} \text{ by the first Partition call that picks as } \textit{pivot} \text{ an element from } Z_{i,j}] \\
&+ \Pr[z_j \text{ is chosen as } \textit{pivot} \text{ by the first Partition call that picks as } \textit{pivot} \text{ an element from } Z_{i,j}] \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \qquad (1)
\end{aligned}$$

since the set $Z_{i,j}$ contains $j - i + 1$ elements.

Finally we obtain

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{i,j} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} = 2\sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\leq 2\sum_{i=1}^{n-1} [(\ln(n-i+1)+1)-1] \leq 2\sum_{i=1}^{n-1} \ln n = O(n \ln n)
\end{aligned}$$

Here we used that $\sum_{i=1}^{k} \frac{1}{i} = H_k$, the k-th harmonic number, and $\ln k \leq H_k \leq \ln k + 1$. This expression also yields a lower bound of $\Omega(n \ln n)$ for $E[X]$, hence $E[X] = \Theta(n \ln n)$.