CS 228: Introduction to Data Structures
Lecture 12
Monday, February 9, 2015

Partitioning

Here is the partition algorithm from last lecture.

```
PARTITION(arr,first,last)
    // Use the last element as the pivot.
    pivot = arr[last]
    i = first - 1
    for j = first to last - 1
        if arr[j] ≤ pivot
            i++
            swap arr[i] and arr[j]
    // Now put pivot in position i+1.
    swap arr[i+1] and arr[last]
    return i + 1
```

Example. The figure on the next page shows the execution of PARTITION on the array

`arr = (4, 9, 1, 5, 2, 3).`

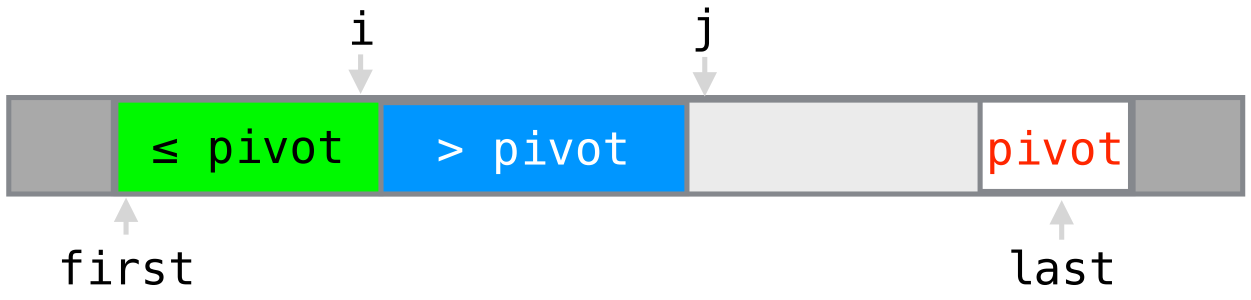
The pivot is shown in red, elements known to be greater than the pivot are blue, elements known to be less than or equal to the pivot are green, and elements that have not yet been considered are grey. Each iteration compares `arr[j]` against the pivot. If `arr[j]` is less than or equal to the pivot, `arr[j]` is swapped with the first blue element. The final step moves the pivot to position `i+1`.

```
i j                 i   j
  4 9 1 5 2 3    ->   4 9 1 5 2 3

i     j               i     j
  4 9 1 5 2 3    ->   1 9 4 5 2 3

  i       j             i       j
  1 9 4 5 2 3    ->   1 2 4 5 9 3

    i       j
  1 2 3 5 9 4
```

PARTITION maintains the following loop invariant.

After each iteration,

- if `first ≤ k ≤ i`, then `arr[k] ≤ pivot`,
- if `i+1 ≤ k ≤ j−1`, then `arr[k] > pivot`, and
- if `k = last`, then `arr[k] = pivot`.



Time complexity of PARTITION. There are n – 1 iterations, where n = last – first+1. Since each iteration takes O(1) time, the time complexity of partition is O(n).

Time Complexity of Quicksort

The running time of QS depends heavily on the sequence of pivots.

Best case: This is achieved when the pivot always splits the array in half. Then, the recursion tree has height O(log n) with O(n) work per level. The total time is O(n log n).

Worst case: This is achieved when one of the subarrays to the side of pivot is empty. Then, the recursion tree has height n, with O(n) work per level, for O(n^2) total. With our implementation of partition, this case occurs when the array is sorted:

```
                1 2 3 4 5
                /       |
         1 2 3 4        5
         /     |
    1 2 3      4
    /   |
 1 2    3
 / |
1  2
```

So, if the array is sorted or nearly sorted, you're better off using insertion sort.

Expected case: A more "typical" case is if array is randomly ordered. Then, odds are good that the pivot will be close to the middle, so each side will contain at most some constant fraction of the elements. Even if the fraction is, say, 0.99n, the running time is O(n log n).

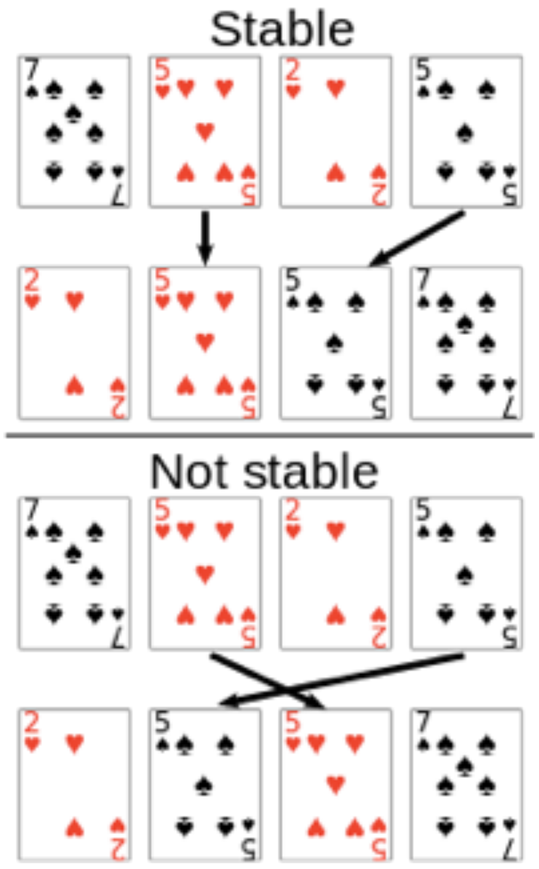
Note. The partition strategy we have studied performs poorly when the data has many duplicates. In particular, if we use it in an array where all elements are *identical*, quicksort will achieve its worst-case running time of $\approx cn^2$, for some constant c. The reason is that the sub-array to the right of the pivot will always be empty (since there are no elements greater than the pivot), while the sub-array to the left will contain n–1 elements. There are variations of partition that handle arrays with duplicates much better than the version we have studied, but we will not cover those variants here.

Pivot Selection in Practice

Using a randomly selected pivot is equivalent to having randomly ordered data. Thus, we can achieve O(n log n) expected time via random pivot selection. Random pivot selection is rather impractical, though. A more common approach is to extract a small sample from the array and use its median as the pivot. For instance, in ***median-of-***

three partitioning picks three array elements (e.g., first, middle, and last) and uses the median of the three as the pivot.

Stability



The things we have to sort are usually more complex than plain numbers. This means that there might be multiple different correctly sorted versions of the original input. For instance, consider the card sorting example on the right[1]. If we sort the cards by rank, there are two possible orderings, depending on how we arrange the two 5 cards. The first ordering preserves the relative ordering between these two cards (hearts before spades); the second one reverses it.

As another example, suppose we have a list of records, each of which consists of the name of a person and that

[1]From http://en.wikipedia.org/wiki/Sorting_algorithm

person’s age. Suppose the records are arranged in alphabetical order by name; e.g.,

(Alice, 18), (Chip, 14), (Dan, 14), (Ellie, 18)

There are four valid ways to sort by age; two of these are

(Chip, 14), (Dan, 14), (Alice, 18), (Ellie, 18)

and

(Dan, 14), (Chip, 14), (Ellie, 18), (Alice, 18)

The, first of these preserves the alphabetical order between Alice and Ellie and between Chip and Dan; the second one does not.

More formally, suppose we are sorting a collection of objects by a certain ***key***. For instance, in the card sorting example, the key was the rank; in the person/age example, the key was the age. A sorting algorithm is ***stable*** if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

Stability can be an important consideration in practice, since data often has a certain underlying structure that we might want to preserve (e.g., alphabetical order). Not all sorting algorithms are stable, although we can always make them stable with some additional work.

Exercise. Which of the sorting algorithms that we have seen are stable? For any algorithm that is not stable, can you show how you can convert it to a stable one?

Generics and Sorting

Here is the Java code for selection sort on an `int` array.

```
public static void selectionSort(int[] arr)
{
  for (int i = 0; i < arr.length - 1; ++i)
  {
    int minIndex = i;
    for (int j = i+1; j < arr.length; ++j)
    {
      if (arr[j] < arr[minIndex])
      {
        minIndex = j;
      }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

If, instead of `ints`, we wanted to sort `Strings` alphabetically, the algorithm would be the same, but the comparisons would be done differently, so that, e.g., `“apple”` would precede `“bear”`. The “<“ operator will not help us here — it is meant for numbers, not strings —

but there is an easy fix: use the `compareTo()` method instead of “<“.

```
public static void
selectionSort(String[] arr)
{
  for (int i = 0; i < arr.length - 1; ++i)
  {
    int minIndex = i;
    for (int j = i+1; j < arr.length; ++j)
    {
      if (arr[j].compareTo(arr[minIndex])
           < 0)
      {
        minIndex = j;
      }
    }
    String temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

This works because Java’s `String` class implements the `Comparable` interface, which consists of objects that have a “natural” ordering. `Comparable` objects have a `compareTo()` method that enables us to determine their order relative to any other object of the same type.

Next time, we'll study the `Comparable` interface, the `Comparator` interface, and generic sorting.