Data Structures

Quicksort

Mr. Khaleifah Al.jada'
Al- Majma'ah Community College

Quicksort

- Quicksort pros:
  - Sorts in place
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice
- Quicksort cons:
  - Sorts $O(n^2)$ in the worst case
  - not stable
    - does not preserve the relative order of elements with equal keys

      Sorting alg stable if 2 records with same key stay in original order.
  - But in practice, it's quick
  - And the worst case doesn't happen often … sorted

Quicksort

- Another divide-and-conquer algorithm:
- *Divide*: $A[p \ldots r]$ is partitioned (rearranged) into two nonempty subarrays $A[p \ldots q\text{-}1]$ and $A[q\text{+}1 \ldots r]$ s.t. each element of $A[p \ldots q\text{-}1]$ is less than or equal to each element of $A[q\text{+}1 \ldots r]$. Index q is computed here, called **pivot**.
- *Conquer*: two subarrays are sorted by recursive calls to quicksort.
- *Combine*: unlike merge sort, no work needed since the subarrays are sorted in place already.

Quicksort

- The basic algorithm to sort an array A consists of the following four easy steps:
  - If the number of elements in A is 0 or 1, then return
  - Pick any element $\boldsymbol{v}$ in A. This is called the ***pivot***
  - Partition $A\text{-}\{\boldsymbol{v}\}$ (the remaining elements in A) into two disjoint groups:
    - $A_1 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \leq \boldsymbol{v}\}$, and
    - $A_2 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \geq \boldsymbol{v}\}$
  - return
    - { quicksort(A_1) followed by $\boldsymbol{v}$ followed by quicksort(A_2)}

Quicksort

- Small instance has $n \leq 1$
  - Every small instance is a sorted instance
- To sort a large instance:
  - select a pivot element from out of the ***n*** elements
- Partition the ***n*** elements into 3 groups left, middle and right
  - The middle group contains only the pivot element
  - All elements in the left group are ≤ pivot
  - All elements in the right group are ≥ pivot
- Sort left and right groups recursively
- Answer is sorted left group, followed by middle group followed by sorted right group

Example

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot

2	5	4	1	3	6	7	9	10	11	8

Sort left and right groups recursively

Quicksort Code

```
Quicksort(A, p, r)
{
     if (p < r)
     {
          q = Partition(A, p, r)
          Quicksort(A, p    , q-1)
          Quicksort(A, q+1  , r)
     }
}
```

- Initial call is **Quicksort**(A, 1, n), where n in the length of A

Partition

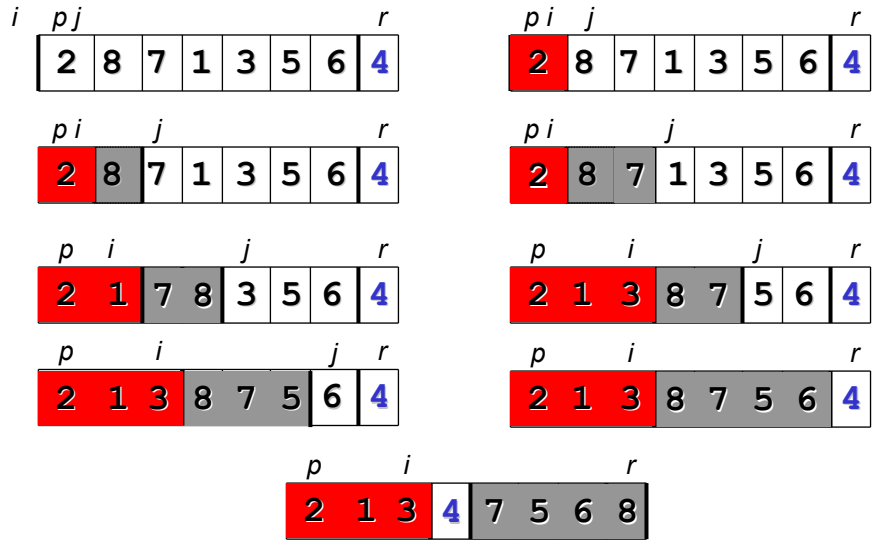
- Clearly, all the action takes place in the `partition()` function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the **index** of the "pivot" element separating the two subarrays

Partition Code

```
Partition(A, p, r)
{
     x = A[r]          // x is pivot
     i = p - 1
      for j = p to r - 1
      {
          do if A[j] <= x
                then
                {
                  i = i + 1
                  exchange A[i] ↔ A[j]
                }
      }
     exchange A[i+1] ↔ A[r]
     return i+1
}
```

partition() runs in O(n) time

Partition Example
$A = \{2, 8, 7, 1, 3, 5, 6, 4\}$



Partition Example Explanation

- Red shaded elements are in the first partition with values $\leq x$ (pivot)
- Gray shaded elements are in the second partition with values $\geq x$ (pivot)
- The unshaded elements have no yet been put in one of the first two partitions
- The final white element is the pivot

Summary: Quicksort

- In worst-case, efficiency is $\Theta(n^2)$
  - But easy to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used
  - Many ways to tune its performance
  - Can be combined effectively
- Various strategies for Partition
  - Some work better if duplicate keys

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	• in-place • slow (good for small inputs)
insertion-sort	$O(n^2)$	• in-place • slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	• in-place, randomized • fastest (good for large inputs)
heap-sort	$O(n \log n)$	• in-place • fast (good for large inputs)
merge-sort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)

Quicksort Code

```
Quicksort(A, p, r)
{
   if (p < r)
   {
      q = Partition(A, p, r);
      Quicksort(A, p, q);
      Quicksort(A, q+1, r);
   }
}
```

Partition Code

```
Partition(A, p, r)
   x = A[p];
   i = p - 1;
   j = r + 1;
   while (TRUE)
      repeat
         j--;
      until A[j] <= x;
      repeat
         i++;
      until A[i] >= x;
      if (i < j)
         Swap(A, i, j);
      else
         return j;
```

Illustrate on
`A = {5, 3, 2, 6, 4, 1, 3, 7};`

What is the running time of `partition()`*?*

Partition Code

```
Partition(A, p, r)
   x = A[p];
   i = p - 1;
   j = r + 1;
   while (TRUE)
      repeat
         j--;
      until A[j] <= x;
      repeat
         i++;
      until A[i] >= x;
      if (i < j)
         Swap(A, i, j);
      else
         return j;
```

`partition()` *runs in O(n) time*