QUICK SORT

Algorithm, Complexity

Quicksort: Intro

- Divide and conquer strategy:
  - Partition array into left and right sub-arrays
    - Choose an element (pivot) of the array
    - Elements in left sub-array are all ≤ pivot
    - Elements in right sub-array are all ≥ pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time
Quicksort: Steps for Sorting

- **QuickSort(S):**
  - If number of elements in S is 0 or 1, return.
  - Pick an element v (pivot) in S.
  - Partition S – {v} into two disjoint subsets,
    - \( S_1 = \{ \text{all values } x \leq v \} \)
    - \( S_2 = \{ \text{all values } x \geq v \} \)
  - Return \{QuickSort(S1), v, QuickSort(S2)\}

Quicksort: Example

If number of elements in S is 0 or 1, return.
Pick an element v (pivot) in S.
Partition S – {v} into two disjoint subsets,
S₁ = \{all values \( x \leq v \)\}
S₂ = \{all values \( x \geq v \)\}
Return \{QuickSort(S₁), v, QuickSort(S₂)\}
Quicksort: Partitioning In Place

- How would you implement partitioning (given a pivot)?
  - How to do it without using an additional array?
- Swap the pivot with the last element
- Start with left and right ends of the array
  - Increment left until a[left] > pivot
  - Decrement right until a[right] < pivot
  - Swap a[left] and a[right]
- Repeat above until left and right cross
- Swap a[left] and pivot (a[n-1])

1 13 24 15 2 26 27 38

Quicksort: Recursive Code

- Sorting an integer array:
  ```java
  private static void quickSort(int[] A, int left, int right) {
      if (left + cutoff <= right) {
          pivot = selectPivot(A, left, right);
          pivotPos = partition(A, left, right, pivot);
          quickSort(A, left, pivotPos -1);
          quickSort(A, pivotPos + 1, right);
      } else insertionSort(A, left, right);
  }
  public static void quickSort(int[] A) {
      quickSort(A, 0, A.length - 1);
  }
  ```
- Look up the partition code in textbook
Quicksort: Run-time Performance

- Let $T(k)$ – run-time to quicksort an array of $k$ elements
  - $T(1) \leq c_0$
  - $T(N) = S(N) + P(N) + T(N_1) + T(N - N_1 - 1)$
    - $S(N) =$ time to select pivot
    - $P(N) =$ time to partition
    - $N_1 =$ size of left sub-array
  - $T(N) \leq c_1 + c_2 N + T(N_1) + T(N - N_1 - 1)$

Quicksort: Pivots and Run-time

- Let $T(k)$ – run-time to quicksort an array of $k$ elements
  - $T(1) \leq c_0$
  - $T(N) \leq c_1 + c_2 N + T(N_1) + T(N - N_1 - 1)$
- Difference from mergesort – $N_1$ may not be $\approx N/2$
- Best pivot selection: sub-array is always split in half
  - $T(N) \leq c_1 + c_2 N + T(N/2) + T(N/2 - 1)$
  - Same as mergesort – $O(N \log N)$
- Worst pivot: one sub-array is empty at each step
  - $T(N) \leq c_1 + c_2 N + T(1) + T(N-1)$
    $\leq c_3 + c_2 N + T(N-1)$
  - Same as insertionSort – $O(N^2)$
- Average case is $O(N \log N)$ – see textbook
Quicksort: Choosing the Pivot

- Element at first (or last) location
  - Popular, but uninformed choice
  - Could be a pretty bad choice if input is not random
    - If input is presorted, or in reverse order – quadratic time!

- Choose the pivot randomly
  - Good choice
  - But random number generators:
    - Could be flawed
    - Are generally expensive to implement

- Median-of-three
  - Best choice of pivot – median of the array
    - Calculating this is expensive!
  - Instead, median of first, middle, and last elements
  - Observed to be a good choice, if data is randomly ordered

Quicksort: Properties


- Run-time
  - Worst-case – $O(N^2)$
  - Best-case – $O(N \log N)$
  - Average-case – $O(N \log N)$ – uses very few comparisons

- Space:
  - $O(\log N) – O(N)$: uses less space than mergesort

- Stability?
  - Not stable – e.g., pivot is reordered with respect to same-value elements

- Not good for small arrays

- Has good in-memory performance
  - Small footprint and good locality
In-class problems

- What is the run-time of quicksort when all keys are equal?
- Given: 1, 2, 3, 4, 5
  - Construct a permutation that is as bad as possible for quicksort (using median-of-three partitioning)