Picking the Pivot

- How would you pick the pivot?
- Strategy 1: Pick the first or last element in **S**
  - Good for a randomly populated array
  - What if input **S** is sorted, or even mostly sorted?
    - All the remaining elements would go into either **S1** or **S2**!
- Strategy 2: Pick the pivot randomly
  - Good in practice if "truly random"
  - Still possible to get some bad choices
  - Requires execution of random number generator

Picking the Pivot (Cont.)

- Strategy 3: Median-of-three Partitioning
  - *Ideally*, the pivot should be the median of input array **S**
    - Median = element in the middle of the sorted sequence
  - Would divide the input into two almost equal partitions
  - Unfortunately, its hard to calculate median quickly, without sorting first!
  - So find the approximate median
    - Pivot = median of the left-most, right-most and center element of the array **S**
    - Solves the problem of sorted input

Picking the Pivot (Cont.)

- Example: Median-of-three Partitioning
  - Let input S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}
  - left=0 and S[left] = 6
  - right=9 and S[right] = 8
  - center = (left+right)/2 = 4 and S[center] = 0
  - Pivot

    = Median of S[left], S[right], and S[center]

    = median of 6, 8, and 0

    = S[left] = 6

Quick Sort vs. Insertion Sort

- For small arrays ($N \leq 20$),
  - Insertion sort is faster than quicksort
- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays
- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small (say for $N < 20$)

QuickSort vs. MergeSort

- Main problem with quicksort:
  - QuickSort may end up dividing the input array into subproblems of size 1 and n-1 in the worst case, at every recursive step (unlike merge sort which always divides into two halves)
    - When can this happen?
    - Leading to $O(n^2)$ performance

  <u>Need to choose pivot wisely (but efficiently)</u>

- MergeSort is typically implemented using a temporary array (for merge step) "not in place"
  - QuickSort can partition the array "in place"

Randomized Quicksort

- When an algorithm has an average case performance and worst case performance that are very different, we can try to minimize the odds of encountering the worst case.

Randomized Quicksort

- Idea: Partition around a random element.
- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- The worst case is determined only by the output of a random-number generator.

Randomized Quicksort

- Randomizing the input
  - With a given set of input numbers, there are very few permutations that produce the worst case performance in Quicksort.
  - But if we pick our pivot randomly, we will rarely get a bad pivot.
  - So, randomly choose a pivot element in A[p..r].
  - For Quicksort, add an initial step to randomize the input array.
  - Running time is now independent of input ordering.

Randomized Partition

```
RANDOMIZED-PARTITION (A, p, r)
    i ← RANDOM (p, r)
    exchange A[r] ↔ A[i]
    return PARTITION (A, p, r)
```

Randomized Quicksort

```
RANDOMIZED-QUICKSORT (A, p, r)
    if p < r
    then q ← RANDOMIZED-PARTITION (A, p, r)
            RANDOMIZED-QUICKSORT (A, p, q-1)
            RANDOMIZED-QUICKSORT (A, q+1, r)
```

Conclusion

- Quicksort is Divide-and-Conquer algorithm.
- In-place sorting
- Quicksort is typically over twice as fast as merge sort.
- Quicksort runs $O(n \log n)$ in the best and average case, but $O(n^2)$ in the worst case.
- Worst case scenarios for Quicksort occur when the array is already sorted, in either ascending or descending order.
- Several strategies to pick the pivot
- We can increase the probability of obtaining average-case performance from Quicksort by using Randomized-partition.