CSE 332: Sorting II

Spring 2016

Announcements

• Midterm: Friday April 29

Sorting: The Big Picture

Simple algorithms: $O(n^2)$

Fancier algorithms: $O(n \log n)$

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$

Handling huge data sets

Sorting algorithms:
- Insertion sort
- Selection sort
- Heap sort
- Merge sort
- Quick sort (avg)
- Bucket sort
- Radix sort
- External sorting

The steps of Quicksort

Quick sort uses a divide and conquer strategy, but does not require the $O(N)$ extra space that MergeSort does.

Here’s the idea for sorting array $S$:

1. Pick an element $v$ in $S$. This is the pivot value.
2. Partition $S$-{v} into two disjoint subsets, $S_1$ and $S_2$ such that:
   - elements in $S_1$ are all $\leq v$
   - elements in $S_2$ are all $\geq v$
3. Return concatenation of QuickSort($S_1$), $v$, QuickSort($S_2$)

Recursion ends when QuickSort() receives an array of length 0 or 1.

Quicksort Example
Pivot Picking and Partitioning

The tricky parts are:

- **Picking the pivot**
  - Goal: pick a pivot value so that $|S_1|$ and $|S_2|$ are roughly equal in size.

- **Partitioning**
  - Preferably in-place
  - Dealing with duplicates

### Picking the Pivot

#### Median of Three Pivot

Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left.

```plaintext
medianOf3Pivot(…)
```

### Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
  - elements in left sub-array are $\leq$ pivot
  - elements in right sub-array are $\geq$ pivot

- Can be done in-place with another “two pointer method”
  - Sounds like mergesort, but here we are partitioning, not sorting...
  - …and we can do it in-place.

### Partitioning In-place

Setup: $i =$ start and $j =$ end of un-partioned elements:

```
0 1 4 9 6 3 5 2 7 0
```

Advance $i$ until element $\geq$ pivot:

```
0 1 4 9 7 3 5 2 6 8
```

Advance $j$ until element $\leq$ pivot:

```
0 1 4 9 7 3 5 2 6 8
```

If $j > i$, then swap:

```
0 1 4 9 7 3 5 2 6 8
```

Advance $i$:

```
0 1 4 9 7 3 5 2 6 8
```

Advance $j$:

```
0 1 4 9 7 3 5 2 6 8
```

Swap:

```
0 1 4 2 7 3 5 5 6 8
```

Advance $i$:

```
0 1 4 2 7 3 5 5 6 8
```

Advance $j$:

```
0 1 4 2 7 3 5 5 6 8
```

If $i > j$, swap in pivot, partition done!

```
0 1 4 2 5 3 7 5 6 8
```

$S_1 \leq$ pivot

$S_2 \geq$ pivot
**Partition Pseudocode**

\[
\text{Partition(A[], left, right)} \{
  v = A[right]; \quad \text{// Assumes pivot value currently at right}
  i = left; \quad \text{// Initialize left side, right side pointers}
  j = right-1;
  \]

  // Do i++, j-- until they cross, swapping values as needed
  while (i) {
    while (A[i] < v) i++;
    while (A[j] > v) j--;
    if (i < j) {
      Swap(A[i], A[j]);
      i++; j--;
    } else
    break;
  }

  Swap(A[i], A[right]); \quad \text{// Swap pivot value into position}
  return i; \quad \text{// Return the final pivot position}
\]

**QuickSort Pseudocode**

Putting the pieces together:

\[
\text{QuickSort(A[], left, right)} \{
  \quad \text{if (left < right)} \{
    \quad \text{medianOf3Pivot(A, left, right)};
    \quad \text{pivotIndex = Partition(A, left+1, right-1)};
    \quad \text{QuickSort(A, left, pivotIndex - 1)};
    \quad \text{QuickSort(A, pivotIndex + 1, right)};
  \}
\}
\]

**QuickSort: Best case complexity**

\[
\text{QuickSort(A[], left, right)} \{
  \quad \text{if (left < right)} \{
    \quad \text{medianOf3Pivot(A, left, right)};
    \quad \text{pivotIndex = Partition(A, left+1, right-1)};
    \quad \text{QuickSort(A, left, pivotIndex - 1)};
    \quad \text{QuickSort(A, pivotIndex + 1, right)};
  \}
\}
\]

**QuickSort: Worst case complexity**

\[
\text{QuickSort(A[], left, right)} \{
  \quad \text{if (left < right)} \{
    \quad \text{medianOf3Pivot(A, left, right)};
    \quad \text{pivotIndex = Partition(A, left+1, right-1)};
    \quad \text{QuickSort(A, left, pivotIndex - 1)};
    \quad \text{QuickSort(A, pivotIndex + 1, right)};
  \}
\}
\]

**QuickSort: Average case complexity**

Turns out to be \(O(n \log n)\).

See Section 7.7.5 for an idea of the proof. \(\text{Don't need to know proof details for this course.}\)

**Many Duplicates?**

An important case to consider is when an array has many duplicates.

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
8 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 0 \\
\end{array}
\]

\[
\text{medianOf3Pivot(...)}
\]

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]
Partitioning with Duplicates

Setup: \( i = \) start and \( j = \) end of un-partioned elements:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( i \) until element \( \geq \) pivot:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( j \) until element \( \leq \) pivot:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

If \( j > i \), then swap:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Partitioning with Duplicates: Take Two

Start \( i = \) start and \( j = \) end of un-partioned elements:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( i \) until element \( > \) pivot (and in bounds):

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( j \) until element \( < \) pivot (and in bounds):

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Finish:

\[
\begin{array}{ccccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Is this better?

Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```java
QuickSort(A[], left, right) {
    if (right - left ≥ CUTOFF) {
        medianOf3Pivot(A, left, right);
        pivotIndex = Partition(A, left+1, right-1);
        QuickSort(A, left, pivotIndex - 1);
        QuickSort(A, pivotIndex + 1, right);
    } else {
        InsertionSort(A, left, right);
    }
}
```

CUTOFF = 16 is reasonable.

Partitioning with Duplicates: Upshot

It's better to stop advancing pointers when elements are equal to pivot, and then just do swaps.

Complexity of quicksort on an array of identical values? Can we do better?

Properties of Quicksort

- \( O(N^2) \) worst case performance, but \( O(N \log N) \) average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- "In-place," but uses auxiliary storage because of recursive calls.
- Stable?
- Used by Java for sorting arrays of primitive types.