CSC 212: Data Structures and Abstractions

Quick Sort

Marco Alvarez
Department of Computer Science and Statistics
University of Rhode Island
Spring 2018

Divide and Conquer

• **Divide** the problem into **smaller** subproblems

• **Conquer** recursively
  • … each subproblem

• **Combine** Solutions

Quick Sort

• **Divide** the array into **two** partitions (subarrays)
  • need to pick a **pivot** and rearrange the elements into two partitions

• **Conquer** **Recursively** each half
  • call Quick Sort on each partition (i.e. solve 2 smaller problems)

• **Combine** Solutions
  • there is no need to combine the solutions

In putting together this issue of Computing in Science & Engineering, we knew three things: it would be difficult to list just 10 algorithms; it would be fun to assemble the authors and read their papers; and, whatever we came up with in the end, it would be controversial. We tried to assemble the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century. Following is our list (here, the list is in chronological order; however, the articles appear in no particular order):

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- Quicksort Algorithm for Sorting
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method
Quick Sort: pseudocode

```c
if (hi <= lo) return;
int p = partition(A, lo, hi);
quicksort(A, lo, p-1);
quicksort(A, p+1, hi);
```

Partition

```
10 12 3 7 4 13 11 9
```

- **pick a pivot** (it can be the first element)

```
4 9 3 7 10 13 11 12
```

- **<= pivot**
- **>= pivot**

Partition: tracing

```
10 1 31 20 3 4 22 15 2 35
```

- **i**
- **j**

- **swap and update i and j**

```
10 1 2 20 3 4 22 15 31 35
```

- **i**
- **j**

- **swap and update i and j**

```
10 1 2 4 3 20 22 15 31 35
```

- **i**
- **j**

- **i and j crossed, time to swap the pivot**

```
3 1 2 4 10 20 22 15 31 35
```
### Partition: do it yourself

| 12 | 1 | 31 | 20 | 10 | 11 | 8 | 2 | 23 | 1 |

### Partition: implementation

```c
int partition(int *A, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    while (1) {
        // while A[i] < pivot, increase i
        while (A[++i] < A[lo]) if (i == hi) break;
        // while A[i] > pivot, decrease j
        while (A[lo] < A[--j]) if (j == lo) break;
        // if i and j cross exit the loop
        if (i >= j) break;
        // swap A[i] and A[j]
        std::swap(A[i], A[j]);
    }
    // swap the pivot with A[j]
    std::swap(A[lo], A[j]);
    // return pivot's position
    return j;
}
```

### Quick Sort: implementation

```c
void r_quicksort(int *A, int lo, int hi) {
    if (hi <= lo) return;
    int p = partition(A, lo, hi);
    r_quicksort(A, lo, p-1);
    r_quicksort(A, p+1, hi);
}
```

```c
void quicksort(int *A, int n, int m) {
    // shuffle the array
    std::random_shuffle(A, A+n);
    // call recursive quicksort
    r_quicksort(A, 0, n-1);
}
```

### Quick Sort: random array

[Image: https://www.toptal.com/developers/sorting-algorithms/quick-sort]
Quick Sort: reversed array

https://www.toptal.com/developers/sorting-algorithms/quick-sort

Analysis of Quick Sort

• **Best-case**
  - almost never happens (array is always partitioned evenly)

\[
T(n) = 2T(n/2) + \Theta(n) \\
= \ldots \\
= \Theta(n \log n)
\]

• **Worst-case**
  - input sorted, reverse order, equal elements

\[
T(n) = T(n - 1) + \Theta(n) + T(0) \\
T(n) = T(n - 1) + \Theta(n) + \Theta(1) \\
= T(n - 1) + \Theta(n) \\
= \ldots \\
= \Theta(n^2)
\]

random shuffle the array (to avoid the worst-case)

• **Average-case**
  - analysis is more complex (assumes distinct elements)
  - faster than merge sort in practice (less data movement)

\[
T(n) = T(9n/10) + T(n/10) + cn \\
= \ldots \\
= \Theta(n \log n)
\]

• Consider a 9-to-1 proportional split
• Even a 99-to-1 split yields same running time
Comments on Quick Sort

- **Properties**
  - It is in-place but **not stable**
  - Benefits substantially from code tuning

- **Improvements**
  - Use insertion sort for small arrays
    - Avoid overhead on small instances (~10 elements)
  - Median of 3 elements
    - Estimate true median by inspecting 3 random elements
  - Three-way partitioning
    - Create three partitions < pivot, == pivot, > pivot

Empirical Analysis

**Running time estimates:**
- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Insertion Sort (N²)</th>
<th>Mergesort (N log N)</th>
<th>Quicksort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td></td>
<td>million</td>
<td>billion</td>
<td>thousand</td>
</tr>
<tr>
<td></td>
<td>billion</td>
<td></td>
<td>million</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>billion</td>
</tr>
<tr>
<td>Home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>18 min</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>0.6 sec</td>
<td>12 min</td>
</tr>
<tr>
<td>Super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

**Lessons**
- Good algorithms are better than supercomputers.
- Great algorithms are better than good ones.