CmSc 250 Intro to Algorithms

Sorting Algorithms: QuickSort

1. Features

- Similar to mergesort - divide-and-conquer recursive algorithm
- One of the fastest sorting algorithms
- Average running time O(NlogN)
- Worst-case running time $O(N^2)$

2. Basic idea

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, re-arranging the entries so that:
    - the pivot is in its proper place.
    - entries smaller than the pivot are to the left of the pivot.
    - entries larger than the pivot are to its right.
3. Recursively apply quicksort to the part of the array that is to the left of the pivot, and to the right part of the array.

Here we don't have the merge step, at the end all the elements are in the proper order.

3. Algorithm

STEP 1. Choosing the pivot

Choosing the pivot is an essential step.
Depending on the pivot the algorithm may run very fast, or in quadric time.:

1. Some fixed element: e.g. the first, the last, the one in the middle

   This is a bad choice - the pivot may turn to be the smallest or the largest element, then one of the partitions will be empty.

2. Randomly chosen (by random generator) - still a bad choice.
3. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.
4. The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

The first element is 8, the middle - 10, the last - 5.
The three elements are sorted: [5, 8, 10] and the middle element is 8. This is the median.

STEP 2. Partitioning

Partitioning is illustrated on the above example.

After finding the pivot the array will look like this:

5, 3, 25, 6, **8,** 17, 1, 2, 18, **10**

1. The first action is to get the pivot out of the way - swap it with the next to the last element

   5, 3, 25, 6, 18, 17, 1, 2, **8, 10**

2. We want larger elements to go to the right and smaller elements to go to the left.

   Two "fingers" are used to scan the elements from left to right and from right to left:

```
[5, 3, 25, 6, 18, 17, 1, 2, 8, 10]
   ^                    ^
   i                    j
```

   Note: we know that the first element is smaller than the pivot, so the first element to be processed is the element to the right. The last two elements are the pivot and an element greater than the pivot, so they are not processed.

   - While **i** is to the left of **j**, we move **i** right, skipping all the elements less than the pivot. If an element is found greater then the pivot, **i** stops
   - While **j** is to the right of **i**, we move **j** left, skipping all the elements greater than the pivot. If an element is found less then the pivot, **j** stops
   - When both **i** and **j** have stopped, the elements are swapped.
   - When **i** and **j** have crossed, no swap is performed, scanning stops, and the element pointed to by **i** is swapped with the pivot .

   In the example the first swapping will be between 25 and 2, the second between 18 and 1.

3. Restore the pivot.

   After restoring the pivot we obtain the following partitioning into three groups:

   [5, 3, 2, 6, 1] [8] [18, 25, 17, 10]

STEP 3. Recursively quicksort the left and the right parts

4. Code

Here is the code, that implements the partitioning.
left points to the first element in the array currently processed, right points to the last element.

```
if( left + 10 <= right)
    {
        int i = left, j = right - 1;
        for ( ; ; )
          {
            while (a[++i] < pivot  ) {}   // move the left finger
            while (pivot  < a[--j] ) {}   // move the right finger

            if (i < j) swap (a[i],a[j]);  // swap
            else  break;                  // break if fingers have crossed
          }
       swap (a[I], a[right-1]);           // restore the pivot
       quicksort ( a, left, i-1);         // call quicksort for the left part
       quicksort (a, i+1, right);         // call quicksort for the left part
     }
else  insertionsort (a, left, right);
```

If the elements are less than 10, quicksort is not very efficient.
Instead insertion sort is used at the last phase of sorting.

Click **here** to see the above example worked out in details

Animation:
http://math.hws.edu/TMCM/java/xSortLab/
In the animation, the leftmost element is chosen as a pivot.

5. Implementation notes

Compare the two versions:

```
A.
        while (a[++i] < pivot) {}
        while (pivot < a[--j]) {}

        if (i < j) swap (a[i], a[j]);
        else break;

B.

        while (a[i] < pivot) {i++;}
        while (pivot < a[j] ) {j--;}

        if (i < j) swap (a[i], a[j]);
        else break;
```

If we have an array of equal elements, the second code will never increment **i** or decrement **j**, and will do infinite swaps. **i** and **j** will never cross.

6. Complexity of Quicksort

Worst-case: $O(N^2)$

> This happens when the pivot is the smallest (or the largest) element.
> Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Best-case O(NlogN) The best case is when the pivot is the median of the array,
and then the left and the right part will have same size.

> There are **logN** partitions, and to obtain each partitions we do **N** comparisons
> (and not more than **N/2** swaps). Hence the complexity is **O(NlogN)**

Average-case - O(NlogN)

Analysis

$$T(N) = T(i) + T(N - i - 1) + cN$$

The time to sort the file is equal to

- the time to sort the left partition with **i** elements, plus
- the time to sort the right partition with **N-i-1** elements, plus
- the time to build the partitions

6. 1. Worst case analysis

The pivot is the smallest element

$$T(N) = T(N-1) + cN, \quad N > 1$$

Telescoping:

$$T(N-1) = T(N-2) + c(N-1)$$

$$T(N-2) = T(N-3) + c(N-2)$$

$$T(N-3) = T(N-4) + c(N-3)$$

$$T(2) = T(1) + c.2$$

Add all equations:

$T(N) + T(N-1) + T(N-2) + \ldots + T(2) =$

$= T(N-1) + T(N-2) + \ldots + T(2) + T(1) + c(N) + c(N-1) + c(N-2) + \ldots + c.2$

$T(N) = T(1) + c(2 + 3 + \ldots + N)$

$T(N) = 1 + c(N(N+1)/2 - 1)$

Therefore **$T(N) = O(N^2)$**

6. 2. Best-case analysis:

The pivot is in the middle

$T(N) = 2T(N/2) + cN$

Divide by N:

$T(N) / N = T(N/2) / (N/2) + c$

Telescoping:

$T(N/2) / (N/2) = T(N/4) / (N/4) + c$

$T(N/4) / (N/4) = T(N/8) / (N/8) + c$

……

$T(2) / 2 = T(1) / (1) + c$

Add all equations:

$T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + \ldots. + T(2) / 2 =$

$= (N/2) / (N/2) + T(N/4) / (N/4) + \ldots + T(1) / (1) + c.logN$

After crossing the equal terms:

$T(N)/N = T(1) + cLogN = 1 + cLogN$

$T(N) = N + NcLogN$

Therefore $T(N) = O(NlogN)$

6. 3. Average case analysis

Similar computations, resulting in $T(N) = O(NlogN)$

The average value of $T(i)$ is $1/N$ times the sum of $T(0)$ through $T(N-1)$

$1/N \Sigma T(j), j = 0 \text{ thru } N-1$

$T(N) = 2/N (\Sigma T(j)) + cN$

Multiply by N

$NT(N) = 2(\Sigma T(j)) + cN*N$

To remove the summation, we rewrite the equation for N-1:

$(N-1)T(N-1) = 2(\Sigma T(j)) + c(N-1)^2, j = 0 \text{ thru } N-2$

and subtract:

$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$

Prepare for telescoping. Rearrange terms, drop the insignificant c:

$NT(N) = (N+1)T(N-1) + 2cN$

Divide by N(N+1):

```
T(N)/(N+1) = T(N-1)/N + 2c/(N+1)
```

Telescope:

```
T(N)/(N+1) = T(N-1)/N + 2c/(N+1)

T(N-1)/(N) = T(N-2)/(N-1)+ 2c/(N)

T(N-2)/(N-1) = T(N-3)/(N-2) + 2c/(N-1)

….

T(2)/3 = T(1)/2 + 2c /3
```

Add the equations and cross equal terms:

```
T(N)/(N+1) = T(1)/2 +2c Σ (1/j), j = 3 to N+1

T(N) = (N+1)(1/2 + 2c Σ(1/j))
```

The sum Σ (1/j), j =3 to N-1, is about LogN

Thus **T(N) = O(NlogN)**

7. Conclusions

Advantages:

- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing). Compare with mergesort: mergesort needs additional memory for merging.

Disadvantages: The worst-case complexity is $O(N^2)$

Applications:

Commercial applications use Quicksort - generally it runs fast, no additional memory, this compensates for the rare occasions when it runs with $O(N^2)$

Never use in applications which require **guaranteed response time**:

- Life-critical (medical monitoring, life support in aircraft and space craft)
- Mission-critical (monitoring and control in industrial and research plants handling dangerous materials, control for aircraft, defense, etc)

unless you assume the worst-case response time.

Comparison with heapsort:

- both algorithms have O(NlogN) complexity
- quicksort runs faster, (does not support a heap tree)
- the speed of quick sort is not guaranteed

Comparison with mergesort:

- mergesort guarantees O(NlogN) time, however it requires additional memory with size N.
- quicksort does not require additional memory, however the speed is not quaranteed
- usually mergesort is not used for main memory sorting, only for external memory sorting.

So far, our best sorting algorithm has **O(*n*log *n*)** performance: can we do any better?

In general, the answer is **no.**

Learning Goals

- Be able to explain how quicksort works and what its complexity is (worst-case, best-case, average case).

- Be able to compare quicksort with heapsort and mergesort.
- Be able to explain the advantages and disadvantages of quicksort, and its applications.

Exam-like questions

1. Briefly describe the basic idea of quicksort.
2. What is the complexity of quicksort?
3. Analyze the worst-case complexity solving the recurrence relation.
4. Analyze the best-case complexity solving the recurrence relation.
5. Compare quicksort with mergesort and heapsort.
6. What are the advantages and disadvantages of quicksort?
7. Which applications are not suitable for quicksort and why?

Back to Contents page

Created by Lydia Sinapova