Data Structures and Algorithm Analysis

Lab 7, Quicksort.

Contents

- Using quicksort with algs4.
- Implement quicksort.

Quicksort with algs4

Normal quicksort:

https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/Quick.html

Quicksort with 3-way partition:

https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/Quick3way.html

Optimized quicksort:

https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/QuickX.html

Using quicksort with algs4

Using the Quick class in algs4 library:

```java
import edu.princeton.cs.algs4.Quick;
import edu.princeton.cs.algs4.StdOut;

public class TestQuickSort {
  public static void main( String[] args ) {

    Integer[] array = new Integer[] { 7, 6, 5, 4, 3, 2, 1,
      10, 9, 8 };

    Quick.sort(array);

    StdOut.println("\nAfter quicksort, the array is:");
    for( Integer i : array ) StdOut.print(" "+i);
    StdOut.println();
  }
}
```

Using quicksort with algs4

The above code should produce the following result:

```
After quicksort, the array is:
 1 2 3 4 5 6 7 8 9 10
```

If you want to use Quick3way or QuickX instead of Quick, just replace the class name.

Using quicksort with algs4

The method we use to do the sorting in Quick is:

```
public static void sort(Comparable[] a)
```

This is the same as mergesort. Again you cannot use int[] directly in the place of Integer[]. They are different in java language.

Implement quicksort

Let's implement our own quicksort.

The same as before we need to define what methods we should have in this class.

```
public class QuickSort {
  public static void sort( int[] array ) {
  }
}
```

Implement quicksort

Same as before we want to write a recursive method, so we define another "sort" method, which specifies the range [left, right] in the signature. The "sort(int[] array)" simply calls that method and specify [0, length-1] as the range. But this time we don't use external memory.

```
private static void sort( int[] array, int low, int high ) {
}
public static void sort( int[] array ) {
  sort(array, 0, array.length-1);
}
```

Implement quicksort

- Select a pivot.
- Divide the array into two part according to the pivot.
- Recursively sort two parts.

```
private static void sort( int[] array, int left, int right )
  {
  int pivot = array[left];
  // divide the array into two parts.
  sort( array, left, ... );
  sort( array, ..., right );
}
```

Let's select the leftmost element as the pivot. We will see the drawback of this arrangement later.

Implement quicksort

Dividing the array into two parts:

```
int pivot = array[right];
int i1 = left-1;
int i2 = right+1;
while( i2-i1 > 1 ) {
  int value = array[i1+1];
  if( value < pivot )
    i1 ++;
  else {
    exchange( array, i1+1, i2-1) ;
    i2--;
  }
}
```

Here we are using two "pointer"s to divide the array into 3 parts and slowly shrink the middle part.

Implement quicksort

Like mergesort we need to write terminate condition for sorting with very small sized array. This will prevent infinite recursion.

```
static void sort( int[] array, int left, int right ) {
  if( right - left < 10 ) {
      insertion(array, left, right);
      return;
  }
  // partition and recursion.
}
```

Implement quick

Finish the method with recursion calls of "sort" and then write you test.

```
...
  sort ( array, left, i1 ) ;
  sort ( array, i2, right );
}

int main() {
  // write your test here
}
```

Test quicksort

Let's do some test, recall the "doubling test" we learned in lab 2.

```
Random rand = new Random();

int[] array = new int[250];
for( int i = 0; i < array.length; i ++ )
  array[i] = rand.nextInt();

double prev = test(array, rand);
for (int n = 250; n > 0; n *= 2) {
  array = new int[n];
  for( int i = 0; i < array.length; i ++ )
    array[i] = rand.nextInt();

  double time = test(array, rand);
  System.out.printf("%7d %7.1f %5.1f\n", n, time, time/prev)
  prev = time;
}
```

Test quicksort

The result looks good.

```
data : 5000 , time : 0.000000 , ratio : NaN
data : 10000 , time : 0.000000 , ratio : NaN
data : 20000 , time : 0.000000 , ratio : NaN
data : 40000 , time : 0.000000 , ratio : NaN
data : 80000 , time : 10.000000 , ratio : Infinity
data : 160000 , time : 19.000000 , ratio : 1.900000
data : 320000 , time : 27.000000 , ratio : 1.421053
data : 640000 , time : 79.000000 , ratio : 2.925926
data : 1280000 , time : 154.000000 , ratio : 1.949367
data : 2560000 , time : 310.000000 , ratio : 2.012987
```

If we were testing merge sort then we can stop here. But since it is quick sort, there's more work to do.

Test quicksort

Let's try it with accending array.

```
...
int[] array = new int[250];
for( int i = 0; i < array.length; i ++ )
  array[i] = i;

double prev = test(array, rand);
for (int n = 250; n > 0; n *= 2) {
  array = new int[n];
  for( int i = 0; i < array.length; i ++ )
    array[i] = i;
...
```

Test quicksort

What happened? Is this the "infinite loop problem" we talked before?

```
Exception in thread "main" java.lang.StackOverflowError
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
  at QuickSort.sort(QuickSort.java:46)
```

Implement quicksort

One way to solve it is to shuffle the array before the sorting. Another way is to randomly select pivot within the interval.

```
int pivot = array[rand.nextInt(right-left+1)+left];
```

```
data : 5000 , time : 12.000000 , ratio : Infinity
data : 10000 , time : 0.000000 , ratio : 0.000000
data : 20000 , time : 0.000000 , ratio : NaN
data : 40000 , time : 0.000000 , ratio : NaN
data : 80000 , time : 5.000000 , ratio : Infinity
data : 160000 , time : 6.000000 , ratio : 1.200000
data : 320000 , time : 21.000000 , ratio : 3.500000
data : 640000 , time : 40.000000 , ratio : 1.904762
data : 1280000 , time : 88.000000 , ratio : 2.200000
data : 2560000 , time : 161.000000 , ratio : 1.829545
data : 5120000 , time : 340.000000 , ratio : 2.111801
```

Test quicksort

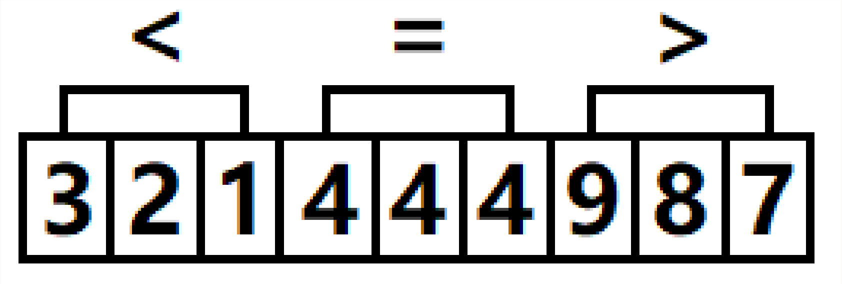
With the pivot being randomly selected, have we finished implementing a quick sort? Try this:

```
int[] array = new int[250];
for( int i = 0; i < array.length; i ++ )
  array[i] = 0;

double prev = test(array, rand);
for (int n = 250; n > 0; n *= 2) {
  array = new int[n];
  for( int i = 0; i < array.length; i ++ )
    array[i] = 0;

  double time = test(array, rand);
  System.out.printf("%7d %7.1f %5.1f\n", n, time, time/prev)
    ;
  prev = time;
}
```

Implement quicksort

One solution is that we need to divide the array into 3 parts instead of 2 parts:

- Elements smaller than pivot.
- Elements equal to pivot.
- Elements bigger than pivot.



Implement quicksort

Let’s rewrite the division part:

```
int i1 = left-1;
int i2 = left-1;
int i3 = right+1;
while( i3-i2 > 1 ) {
  int value = array[i2+1];
  if( value < pivot ) {
    exchange(array, i1+1, i2+1);
    i1++;
    i2++;
  } else if( value > pivot ) {
    exchange(array, i2+1, i3-1);
    i3--;
  } else
    i2++;
}
```

Exercise: ignore small subarrays

Finish execise 2.3.27 in "Algorithms FOURTH EDITION":

Run experiments to compare the following strategy for dealing with small subarrays:

Simply ignore the small subarrays in quicksort, then run a single insertion sort after the quicksort completes.