CS 112 – Introduction to Computing II
Wayne Snyder
Computer Science Department
Boston University

Today
Recursive Sorting Methods and their Complexity:
- Mergesort
- Quicksort
Conclusions on sorting algorithms and complexity

Next Time:
- Introduction to Linked Lists: Reference types on steroids!
- Iterative algorithms on Linked Lists
(Reading: On the web site)

Recursive Sorting: Merge Sort

**Merge Sort** is a relative simple application of recursive (Divide and Conquer) reasoning to the problem of sorting:

1. Divide into two smaller subproblems:
   - Subproblem 1
   - Subproblem 2

2. Solve subproblems recursively:
   - Subproblem 1 solution
   - Subproblem 2 solution

3. Put the (sub)solutions together into a solution:
Merge Sort

Merge Sort is a relative simple application of bottom up recursive (Divide and Conquer) reasoning to the problem of sorting:

Base Case: If the list is empty or has only one element, stop;
Recursive Case: If the list has 2 or more elements, divide in half (best you can), sort each separately, and then merge the two sorted lists into one sorted list:

4 8 6 1 7 2 3 5
divide: 4 8 6 1 7 2 3 5
conquer: 1 4 6 8 2 3 5 7
merge: 1 2 3 4 5 6 7 8

Complexity of Merge Sort

Divide and Conquer
Sorting: Merge Sort

private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    // copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    // merge back to a[]
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) a[k] = aux[j++]; // left side exhausted
        else if (j > hi) a[k] = aux[i++]; // right side exhausted
        else if (less(aux[j], aux[i])) a[k] = aux[j++]; // smallest on right side
        else a[k] = aux[i++]; // minimal on left side
    }
}
// mergesort a[lo..hi] using auxiliary array aux[lo..hi]
private static void mergesort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void mergesort(int[] a) {
    int[] aux = new int[a.length];
    mergesort(a, aux, 0, a.length - 1);
}

Complexity of Merge Sort

Let's count the number of comparisons (calls to less)

Observe that less is called in only one place, in merge, so we start by thinking about what happens when we merge two ordered lists.

What is the best thing that can happen when merging two ordered lists?

All the elements in one list are less than the elements in the other list, e.g., in an already ordered list:

1 2 3 4 5 6 7 8

How many comparisons?
Complexity of Merge Sort

Merge Sort doesn’t use exchanges, so let’s count the number of comparisons (less); since you never move data items without comparing them, this is sufficient for $\Theta(\ldots)$.

Observe that less is called in only one place, in merge.

What is the best thing that can happen when merging two ordered lists?

All the elements in one list are less than the elements in the other list, e.g., in an already ordered list:

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

How many comparisons? 4 (in general: $\Theta(N)$ for $N$ elements)

What is the worst thing that can happen when merging two ordered lists?

The rightmost elements in each list are the two largest elements: and the last comparison must compare these two:

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

So every element except for the largest has to be compared before moving down.

How many comparisons? 7 (in general: $N-1$ or $\Theta(N)$ for $N$ elements)

Punchline: Merge takes linear time: $\Theta(N)$
Complexity of Merge Sort

Now: how many times is Merge called?

Or:

How many times can you divide a list of size $N$ in half when you Divide before Conquering?

$N$  $N/2$  $N/4$  .....  4  2  1

Let’s write it the other way:

$\begin{align*}
1 & \quad 2 & \quad 4 & \quad \ldots & \quad N \\
2^0 & \quad 2^1 & \quad 2^2 & \quad \ldots & \quad 2^{\log(N)} = N
\end{align*}$

Answer: $\Theta \left( \log_2 N \right)$

Complexity of Merge Sort

Merge takes $\Theta \left( N \right)$ comparisons in all cases.
You can divide the list of size $N$ in half $\Theta \left( \log_2 N \right)$ times.

Conclusion: Mergesort takes $\Theta \left( N \cdot \log_2 N \right)$ comparisons in all cases:

<table>
<thead>
<tr>
<th>Subproblem size</th>
<th>Number of Subproblems</th>
<th>Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta \left( \log_2 N \right)$</td>
<td>$N/2$</td>
<td>$\Theta \left( N/2 \cdot 2 \right) = \Theta \left( N \right)$</td>
</tr>
<tr>
<td>$\Theta \left( \log_2 N \right)$</td>
<td>$2$</td>
<td>$\Theta \left( 2 \cdot N/2 \right) = \Theta \left( N \right)$</td>
</tr>
<tr>
<td>$\Theta \left( \log_2 N \right)$</td>
<td>$1$</td>
<td>$\Theta \left( 1 \cdot N \right) = \Theta \left( N \right)$</td>
</tr>
</tbody>
</table>
Merge Sort vs. Quick Sort

**Merge Sort** is a relatively simple application of recursive (Divide and Conquer) reasoning to the problem of sorting. Note that all the work is essentially done by the Combine step by Merging:

- Divide into two smaller subproblems:
- Solve subproblems recursively:
- Put the (sub)solutions together into a solution:

- Divide step is trivial, just separate the two halves! $\Theta(1)$
- Combining two subproblems involves Merge, which is $\Theta(N)$

But could we do it the other way around? You could divide the problem into two pieces, solve them separately, and then simply concatenate. All the work would essentially be done by the Divide step:

- Divide into two smaller subproblems:
- Solve subproblems recursively:
- Put the (sub)solutions together into a solution:

- Divide step is difficult, say $\Theta(N)$
- Combining two subproblems is trivial, just put the two solutions together: $\Theta(1)$
Quicksort

The Quicksort algorithm does just this: it divides the problem in such a way that nothing needs to be done to combine the solutions at the end except to simply concatenate the sub-solutions. This can be done in place, with one array. The process is called Partitioning:

<table>
<thead>
<tr>
<th>List of N numbers in some random order….</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest N / 2 numbers</td>
<td>Largest N / 2 numbers</td>
</tr>
</tbody>
</table>

Note: It does not matter what order the numbers are in after the partition step, as long as the smallest half of the list is to the left, and the largest half is to the right….

Quicksort

Let's look at a concrete example:

5 3 1 6 2 7 8 4
Let's look at an example:

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

Partition Step

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

Notice that each number is now on the correct side of the partition wall; after this step, no number will cross the wall. Each is closer to its final position.

The order of the numbers inside a partition does not matter, as long as each is in the correct partition.

If we repeat this process recursively, we will put all the elements into the correct place.....

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

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

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

```
1 2 3 4 5 6 7 8
```
One way to think about this is that each number is moving closer and closer to its correct location in the sorted array, into the correct half, then the correct fourth, then eighth, etc.

5 3 1 6 2 7 8 4

4 1 3 2 8 5 7 6

1 2 4 3 6 5 8 7

1 2 3 4 5 6 7 8

Preview of Quicksort complexity: Each number can move only log(N) times, and there are N numbers... smells like N * log(N)...
How to implement Partition?

1. Choose a pivot value (we’ll use the leftmost element)
2. For the remaining elements, move all which are ≤ pivot to left, and all ≥ pivot to the right, then move the pivot between them.

**Note:**
- We’ve done the partition, and also put the pivot in exactly the right place!

---

How to implement Partition on an array A?

1. Choose a pivot value V (we’ll use the leftmost element)

```
5 3 1 6 2 7 4 8
V
```
How to implement Partition on an array A?
1. Choose a pivot value V (we’ll use the leftmost element)
2. Set a pointers i and j to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);

3. While A[i] ≤ V, move i to the right, since these A[i] are in the correct partition;
How to implement Partition on an array A?

1. Choose a pivot value $V$ (we'll use the leftmost element)
2. Set a pointers $i$ and $j$ to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);
3. While $A[i] \leq V$, move $i$ to the right, since these $A[i]$ are in the correct partition;

\[
\begin{array}{ccccccccc}
5 & 3 & 1 & 6 & 2 & 7 & 4 & 8 \\
V & i & \rightarrow & i & j
\end{array}
\]

Left Partition $\leq 5$

\[
\begin{array}{ccccccccc}
5 & 3 & 1 & 6 & 2 & 7 & 4 & 8 \\
V & i & & & j
\end{array}
\]
How to implement Partition on an array A?

1. Choose a pivot value V (we’ll use the leftmost element)
2. Set a pointers i and j to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);
3. While A[i] ≤ V, move i to the right, since these A[i] are in the correct partition; stop when A[i] > V or i crosses j (i > j);
4. Do the same from the right: while if A[j] ≥ V, move j to the left, since these are also correct; stop when A[i] > V or i crosses j (i > j);
How to implement Partition on an array A?
1. Choose a pivot value V (we'll use the leftmost element)
2. Set a pointers i and j to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);
3. While A[i] ≤ V, move i to the right, since these A[i] are in the correct partition; stop when A[i] > V or i crosses j (i > j);
4. Do the same from the right: while If A[j] ≥ V, move j to the left, since these are also correct; stop when A[i] > V or i crosses j (i > j);
5. Now A[i] and A[j] are clearly in the wrong partitions, so swap them;

Left Partition ≤ 5
5 3 1 6 2 7 4 8

V i j

Right Partition ≥ 5

5 3 1 6 2 7 4 8

V i j
How to implement Partition on an array A?
1. Choose a pivot value \( V \) (we’ll use the leftmost element)
2. Set a pointers \( i \) and \( j \) to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);
3. While \( A[i] \leq V \), move \( i \) to the right, since these \( A[i] \) are in the correct partition; stop when \( A[i] > V \) or \( i \) crosses \( j \) (\( i > j \));
4. Do the same from the right: while if \( A[j] \geq V \), move \( j \) to the left, since these are also correct; stop when \( A[i] > V \) or \( i \) crosses \( j \) (\( i > j \));
5. Now \( A[i] \) and \( A[j] \) are clearly in the wrong partitions, so swap them;
6. \( A[i] \) and \( A[j] \) are now correct, so move \( i \) to the right and \( j \) to the left and go back to step 3;
Quicksort

How to implement Partition on an array A?

1. Choose a pivot value V (we’ll use the leftmost element)
2. Set a pointers i and j to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross (c.f., Binary Search!);
3. While A[i] ≤ V, move i to the right, since these A[i] are in the correct partition; stop when A[i] > V or i crosses j (i > j);
4. Do the same from the right: while If A[j] ≥ V, move j to the left, since these are also correct; stop when A[i] > V or i crosses j (i > j);
5. Now A[i] and A[j] are clearly in the wrong partitions, so swap them;
6. A[i] and A[j] are now correct, so move i to the right and j to the left and go back to step 3;

```
5 3 1 4 2 7 6 8

Left Partition ≤ 5

Right Partition ≥ 5

V j ← i
```
How to implement Partition on an array A?

1. Choose a pivot value V (we’ll use the leftmost element)
2. Set a pointers i and j to the left- and right-most elements of the remaining numbers; these will move towards each other until they cross;
3. While A[i] ≤ V, move i to the right, since these A[i] are in the correct partition; stop when A[i] > V or i crosses j (i > j); if i crossed j go to step 7;
4. Do the same from the right: while if A[j] ≥ V, move j to the left, since these are also correct; stop when A[i] > V or i crosses j (i > j); if j crossed i, go to step 7;
5. Now A[i] and A[j] are clearly in the wrong partitions, so swap them;
6. A[i] and A[j] are now correct, so move i to the right and j to the left and go back to step 3;
7. Exchange V and A[j] and stop, done!

To Quicksort, just apply partition recursively! Stop when subproblems are of length 0 or 1.
Example of Quicksort:

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>1</th>
<th>6</th>
<th>2</th>
<th>7</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>


Example of Quicksort:

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>1</th>
<th>6</th>
<th>2</th>
<th>7</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

V

i

j

1 ≤ 3? ✔️
### Quicksort

#### Example of Quicksort:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

\[ V \quad i \quad j \]

\[4 \leq 3 \ ? \ x\]
\[2 \geq 3 \ ? \ x\]

Swap and move \(i\) and \(j\)
# Quicksort

## Example of Quicksort:

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>1</th>
<th>6</th>
<th>2</th>
<th>7</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Vi

| 2 | 1 | 3 | 4 | 5 | 7 | 6 | 8 |

V i j
### Example of Quicksort:

```
5 3 1 6 2 7 8 4
3 1 4 2 5 7 6 8
2 1 3 4 5 7 6 8
2 1 3 4 5 7 6 8
V  j  i
```

### Example of Quicksort:

```
5 3 1 6 2 7 8 4
3 1 4 2 5 7 6 8
2 1 3 4 5 7 6 8
1 2 3 4 5 7 6 8
```
### Example of Quicksort:

1. **Initial Array:**
   
   | 5 | 3 | 1 | 6 | 2 | 7 | 8 | 4 |

2. **After Partitioning:**
   
   | 3 | 1 | 4 | 2 | 5 | 7 | 6 | 8 |

3. **After Repeated Partitioning:**
   
   | 2 | 1 | 3 | 4 | 5 | 7 | 6 | 8 |

4. **Final Sorted Array:**
   
   | 1 | 2 | 3 | 4 | 5 | 7 | 6 | 8 |

Values: V, i, j
### Quicksort

**Example of Quicksort:**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>V</td>
<td>j</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example of Quicksort:**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick Sort

```java
private static void quickSort(int[] A) {
    qsHelper(A, 0, A.length - 1);
}

// quicksort the subarray from A[lo] to A[hi]
private static void qsHelper(int[] A, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(A, lo, hi);
    qsHelper(A, lo, j-1);
    qsHelper(A, j+1, hi);
}

// partition the subarray A[lo..hi] and return location j of pivot
private static int partition(int[] A, int lo, int hi) {
    int i = lo+1;
    int j = hi;
    int v = A[lo];
    while (i <= j) {
        while (i < A.length && less(A[i], v))
            ++i;
        while (less(v, A[j]))
            --j;
        if (i > j)
            break;
        else {
            swap(A, i, j);
            ++i;
            --j;
        }
    }
    swap(A, lo, j);  // put pivot v at A[j]
    return j;
}
```

Quick Sort: Complexity

We have guessed that Quicksort will be $N \log(N)$ because we keep breaking each problem into partitions which are less than $\frac{1}{2}$ the original size:

Preview of Quicksort complexity: Each number can move only $\log(N)$ times, and there are $N$ numbers.... smells like $N \times \log(N)$...
Quick Sort: Complexity

But remember we are counting comparisons, so let's consider how many comparisons Partition takes for a list of length $N$.

Note that after every comparison, we either move $i$ or $j$, or swap two numbers; so each comparison puts a number in the correct partition, however, when the $i$ and $j$ cross we may have compared $A[i]$ and $A[j]$ each an extra time; so we will always do at most $N+1$ comparisons, or $\Theta(N)$.

<table>
<thead>
<tr>
<th>Left Partition</th>
<th>$\leq 5$</th>
<th>Right Partition</th>
<th>$\geq 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$i$</td>
<td>8</td>
<td>$j$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V$</td>
<td></td>
<td>$V$</td>
<td></td>
</tr>
</tbody>
</table>

Quick Sort: Complexity

How many comparisons overall?

If we are lucky, the pivot is the median (middle number) of the list, and the problem breaks into two subproblems of $\leq N/2$:

List of length $N$

Partition Step

Numbers $\leq V$ | $V$ | Numbers $\geq V$

$\leq N/2$ numbers | $\leq N/2$ numbers
Quick Sort: Complexity

How many comparisons overall?

If we are lucky, the pivot is the median (middle number) of the list, and the problem breaks into two subproblems of $\leq N/2$;

So we can divide into equal subproblems at most $\log(N)$ times, and the complexity is $\Theta(N \cdot \log(N))$, right???

Quick Sort: Complexity

Is Quicksort really $\Theta(N \cdot \log(N))$ ????

Well, no.... what if we are NOT lucky when we pick the pivot, and we choose the worst possible pivot, which is the smallest or largest number:
And then we continue to have pivots (the leftmost number) which are either the largest or smallest in the subproblem.... suppose we always get the smallest number as pivot......

Size of Subproblem

<table>
<thead>
<tr>
<th>N</th>
<th>N - 1</th>
<th>N - 2</th>
<th>N - 3</th>
<th>N - 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

So., the worst case for Quicksort is $\Theta(N^2)$

## Sorting: Conclusions on Time Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case Input</th>
<th>Worst-case Time</th>
<th>Best-case Input</th>
<th>Best-case Time</th>
<th>Average-case Input</th>
<th>Average-case Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>Any!</td>
<td>$\Theta(N^2)$</td>
<td>Any!</td>
<td>$\Theta(N^2)$</td>
<td>Any!</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Reverse Sorted List</td>
<td>$\Theta(N^2)$</td>
<td>Already Sorted List</td>
<td>$\Theta(N)$</td>
<td>Random List</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>Complicated!</td>
<td>$\Theta(N \log(N))$</td>
<td>Already Sorted List</td>
<td>$\Theta(N \log(N))$</td>
<td>Random List</td>
<td>$\Theta(N \log(N))$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>Already sorted or reverse sorted.</td>
<td>$\Theta(N^2)$</td>
<td>Pivot is always the median.</td>
<td>$\Theta(N \log(N))$</td>
<td>Random List</td>
<td>$\Theta(N \log(N))$</td>
</tr>
</tbody>
</table>
Comparing Sorting Algorithms

![Graph comparing sorting algorithms](image)

Comparing Sorting Algorithms

![Graph comparing sorting algorithms](image)

Computer Science

10/4/16
Comparing Sorting Algorithms

![Comparison of Inversion Sort and Merge Sort](image)

Comparing Sorting Algorithms

![Comparison of Merge Sort and Inversion Sort](image)
Comparing Sorting Algorithms

![Graph comparing sorting algorithms](image)

1. Insertion Sort
2. Selection Sort
3. Merge Sort
4. Quick Sort

Comparing Sorting Algorithms

![Graph comparing sorting algorithms](image)

1. Insertion Sort
2. Selection Sort
3. Merge Sort
4. Quick Sort
Timing Java Code

But is counting comparisons the best way to analyze algorithms? What about how much TIME they take??

This turns out to be a complicated question, because the actual time depends on many, many factors:
- How fast is your processor? Do you have more than 1 processor?
- How many other processes are running? (Example: the Java garbage collector!)
- How much memory do you have? Does this affect really big inputs?
- What operating system?
- Etc., etc., etc.

To do this right, you have to specify ALL these parameters, and run a standard platform with standard benchmarks; this is in fact done when testing new processors.

But assuming we are running two different algorithms on the same platform, we should be able to get some interesting results. Let’s think about how to time Java code…..
Here is a simple way to time a region of Java code:

```java
long startTime = System.currentTimeMillis();
// some code you want to time
long endTime = System.currentTimeMillis();
System.out.println("Total execution time: "+ (endTime – startTime));
```

To get more precision, you can do the code 100000 times, then divide by 100000, etc.

Here is a sample of what I wrote to time our sorting algorithms:

```java
for(int i = 5; i <= 200 ; i+=5){
    int[][] a = new int[i][100000][0];
    for(int j = 0; j < 100000; ++j)
        a[j] = genRandomArray(i);
    long startTime = System.currentTimeMillis();
    for(int j = 0; j < 100000; ++j)
        selectionSort(a[j]); // code to be timed goes here
    long endTime = System.currentTimeMillis();
    System.out.println(i + " " + (endTime – startTime));
}
```
Timing Java Code: Average Case for Actual Time

[Graph showing the performance of different sorting algorithms (Selection Sort, Insertion Sort, Merge Sort, Quick Sort) over a range of data sizes.]

Timing Java Code: Average Case for Actual Time

[Graph showing the performance of different sorting algorithms (Selection Sort, Insertion Sort, Merge Sort, Quick Sort) over a range of data sizes.]
Timing Java Code