CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

Recursive sorting: Mergesort and Quicksort

Next Time:

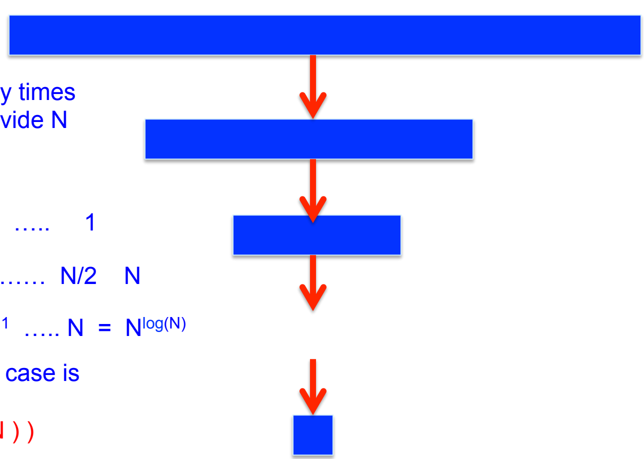
Object-Oriented Design and Java program structure;

Classes: static and non-static

Organizing a program into separate files

Recursive Sorting: Divide and Conquer

The strategy of dividing a problem in half that we used in binary search is called Divide and Conquer and is fundamental to creating more efficient algorithm with a **logarithmic** running time. . It is often amenable to a recursive solution, where each recursive call is on a problem half as big.



Size

N

N/2

N/4

……

1

How many times can we divide N in half?

N N/ 2 ….. 1

Or: 1 2 …… N/2 N

Or: 2^0 2^1 ….. N = $N^{\log(N)}$

So, worst case is

$\Theta(\log_2(N))$

Recursive Sorting: Merge Sort

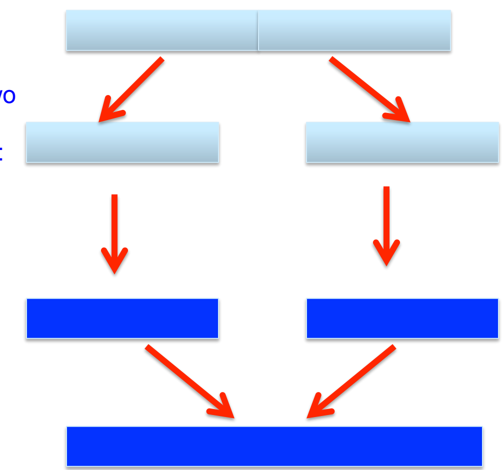
Computer Science

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

Divide into two smaller subproblems:

Solve subproblems recursively:

Put the (sub)solutions together into a solution:



3

Merge Sort

Computer Science

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: **(recursive calls)**

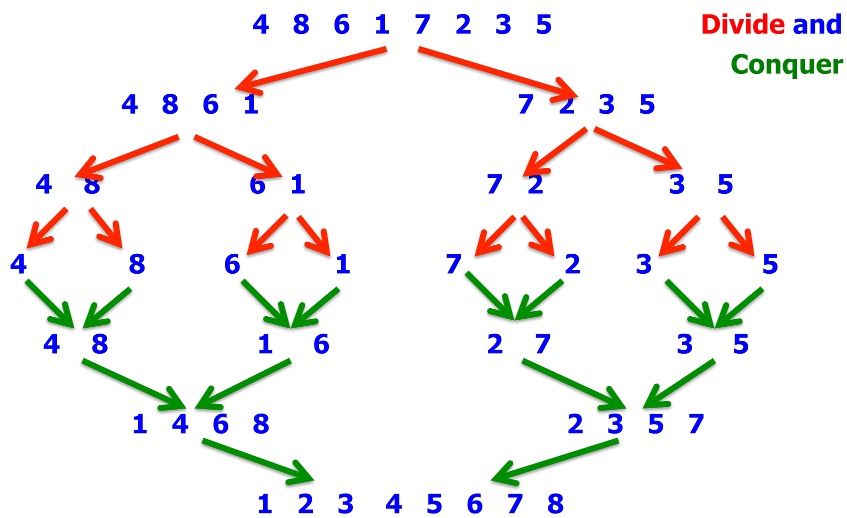
1 4 6 8 **2 3 5 7**

merge: **1 2 3 4 5 6 7 8**

4

Complexity of Merge Sort

Computer Science



5

Sorting: Merge Sort

Computer Science

```
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 exhaused
        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);
}
```

6

Complexity of Merge Sort

Computer Science

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?

7

Complexity of Merge Sort

Computer Science

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 $\Theta(....)$.

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:

1 2 3 4 5 6 7 8

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

8

Complexity of Merge Sort

Computer Science

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:

1 2 3 7 5 6 4 8

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

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

Punchline: Merge takes linear time: Θ(N)

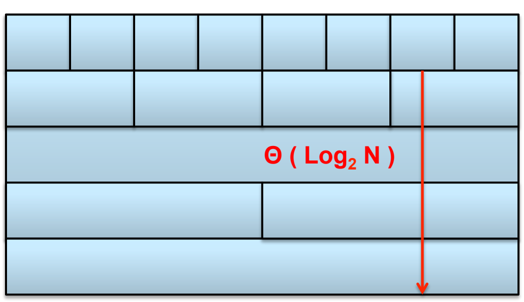
Complexity of Merge Sort

Computer Science

Merge takes Θ(N) comparisons in all cases.

You can divide the list of size N in half Θ (Log_2 N) times.

Conclusion: Mergesort takes Θ (N * Log_2 N) comparisons in all cases:



Subproblem size	Number of Subproblems	Number of Comparisons
(Initial Problem)		
2	N/2	Θ(N/2 * 2) = Θ(N)
N/2	2	Θ(2 * N/2) = Θ(N)
N	1	Θ(1 * N) = Θ(N)

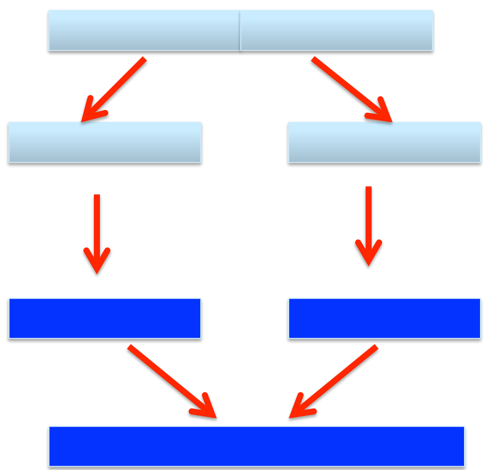
Merge Sort vs. Quick Sort

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 step is difficult, say Θ(N)

Divide into two smaller subproblems:

Solve subproblems recursively:



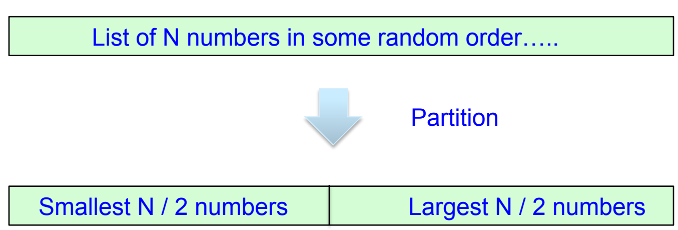
Put the (sub)solutions together into a solution:

Combining two subproblems is trivial, just put the two solutions together: Θ(1)

11

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:



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

Computer Science

Let's look at a concrete example:

5 3 1 6 2 7 8 4

Quicksort

Computer Science

Let's look at an example:

5 3 1 6 2 7 8 4

Partition Step

4 1 3 2 | 8 5 7 6

Wall

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.

Quicksort

Computer Science

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

Quicksort

Computer Science

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

Quicksort

Computer Science

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 ½ of the list, then the correct ¼ , then 1/8 , etc.

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

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

Quicksort

Computer Science

How to implement Partition?

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

| List of N unordered numbers |

Pick leftmost number as pivot

| V | List of N-1 unordered numbers |

Partition Step

| V | Numbers ≤ V n | Numbers ≥ V |

Swap pivot with rightmost element of left partition

| n Numbers <= V | V | Numbers >= V |

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

Quicksort

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							

Quicksort

Computer Science

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!);

5	3	1	6	2	7	4	8
V	i						j

Quicksort

Computer Science

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;

3 ≤ 5 ? ✔

5 3 1 6 2 7 4 8

V i ⟶ i j

Quicksort

Computer Science

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;

1 ≤ 5 ? ✔

5 3 1 6 2 7 4 8

V i ⟶ i j

Quicksort

Computer Science

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;

Left Partition ≤ 5

5 3 1 6 2 7 4 8

V i j

Quicksort

Computer Science

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);

Left Partition ≤ 5

6 ≤ 5 ? ✗

5 3 1 6 2 7 4 8

V i j

Quicksort

Computer Science

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);

Left Partition ≤ 5

8 ≥ 5 ? ✔

5 3 1 6 2 7 4 8

V i j ← j

Quicksort

Computer Science

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);

Left Partition ≤ 5

4 ≥ 5 ? ✗

5 3 1 6 2 7 4 8

V i j ← j

Quicksort

Computer Science

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 Right Partition ≥ 5

5 3 1 6 2 7 4 8

V i j

Quicksort

Computer Science

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;

Left Partition ≤ 5 Right Partition ≥ 5

5 3 1 4 2 7 6 8

V i → i j ← j

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;

Left Partition ≤ 5 Right Partition ≥ 5

5 3 1 4 2 7 6 8

V i → i j

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;

Left Partition ≤ 5 Right Partition ≥ 5

5 3 1 4 2 7 6 8

V j ← i j

Quicksort

Computer Science

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;

Left Partition ≤ 5 | Right Partition ≥ 5

5 3 1 4 2 | 7 6 8

V ... j | i

Quicksort

Computer Science

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!

Left Partition ≤ 5 | Right Partition ≥ 5

5 3 1 4 2 | 7 6 8

V ... j | i

Quicksort

Computer Science

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! Pivot in exactly the right spot, all other numbers partitioned.

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

Now partition this subproblem. And this subproblem

$$\underbrace{3 \quad 1 \quad 4 \quad 2}\quad 5 \quad \underbrace{7 \quad 6 \quad 8}$$

↑ In the correct location

Quicksort

Computer Science

Example of Quicksort:

5 3 1 6 2 7 8 4

3 1 4 2 | 5 | 7 6 8

Quicksort

Computer Science

Example of Quicksort:

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

1 ≤ 3 ? ✔

Quicksort

Computer Science

Example of Quicksort:

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

4 ≤ 3 ? ✗
2 ≥ 3 ? ✗

Quicksort

Example of Quicksort:

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

Swap and
move i and j

Quicksort

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

Quicksort

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	i j						

Quicksort

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					

Quicksort

Computer Science

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

Quicksort

Computer Science

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
					V	i	j

Quicksort

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
					V		i j

Quicksort

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
					V	j	i

Quicksort

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
1	2	3	4	5	6	7	8

Quick Sort


```
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]
                        // now, A[lo .. j-1] <= A[j] <= A[j+1 .. hi]
  return j;
}
```

46

Quick Sort: Complexity

Computer Science

We have guessed that Quicksort will be N*log(N) because we keep breaking each problem into partitions which are about ½ the original size:

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

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
1	2	3	4	5	6	7	8

Quick Sort: Complexity

Computer Science

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 Θ(N).

Left Partition ≤ 5   Right Partition ≥ 5

5 3 1 4 2 7 6 8

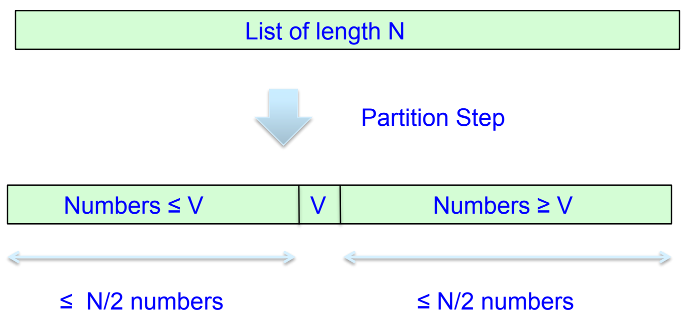
V   i → i   j ← j

Quick Sort: Complexity

Computer Science

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 ≤ N/2:



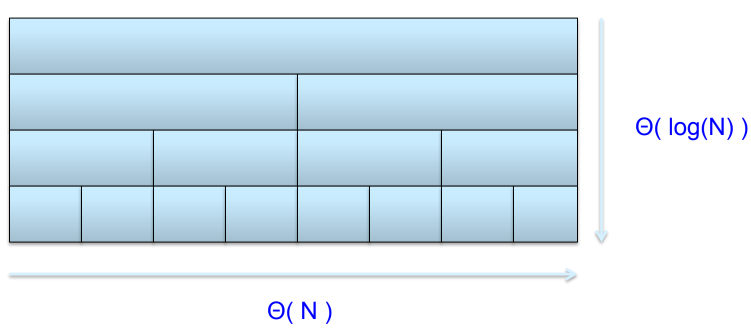
Quick Sort: Complexity

Computer Science

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 ≤ N/2;

So we can divide into equal subproblems at most log(N) times, and the complexity is Θ(N * log(N)), right????

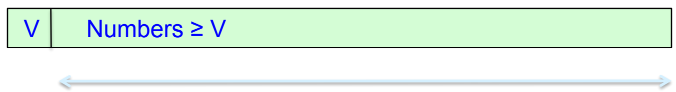


Quick Sort: Complexity

Computer Science

Is Quicksort really Θ(N * 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:



No numbers ≤ V !

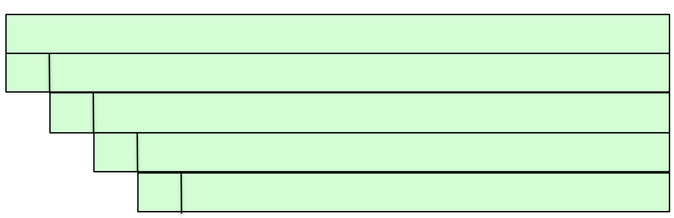
N – 1 numbers ≥ V

Quick Sort: Complexity

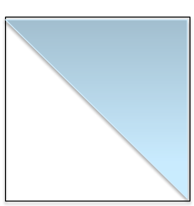
Computer Science

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……

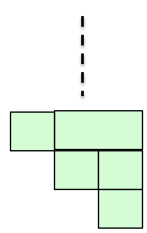
Size of Subproblem



N
N - 1
N - 2
N - 3
N - 4



So.. the worst case for Quicksort is Θ(N^2)



2
1

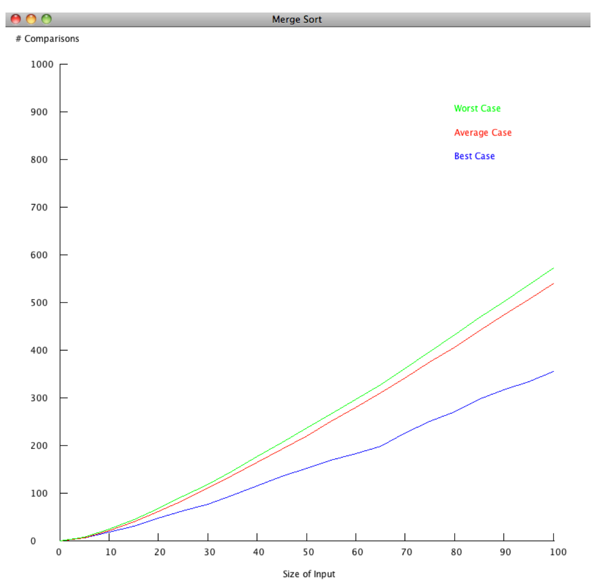
Sorting: Conclusions on Time Complexity

Computer Science

Algorithm	Worst-case Input	Worst-case Time	Best-case Input	Best-case Time	Average-case Input	Average-case Time
Selection Sort	Any!	$\Theta(N^2)$	Any!	$\Theta(N^2)$	Any!	$\Theta(N^2)$
Insertion Sort	Reverse Sorted List	$\Theta(N^2)$	Already Sorted List	$\Theta(N)$	Random List	$\Theta(N^2)$
Mergesort	*Complicated!*	$\Theta(N^* \log(N))$	Already Sorted List	$\Theta(N^* \log(N))$	Random List	$\Theta(N^* \log(N))$
Quicksort	Already sorted or reverse sorted.	$\Theta(N^2)$	Pivot is always the median.	$\Theta(N^* \log(N))$	Random List	$\Theta(N^* \log(N))$

Comparing Sorting Algorithms

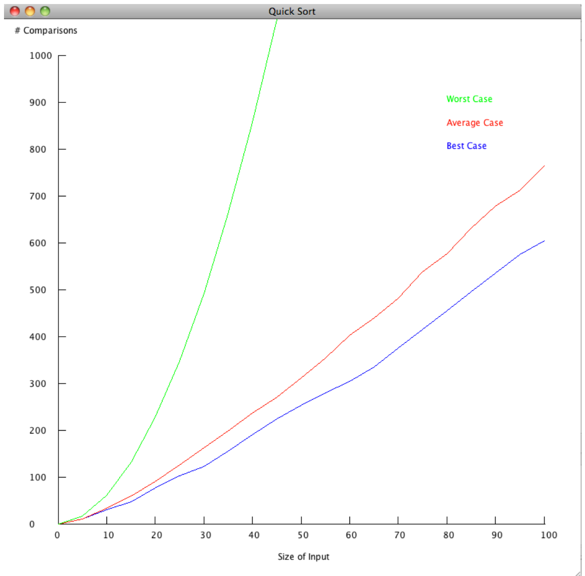
Computer Science



54

Comparing Sorting Algorithms

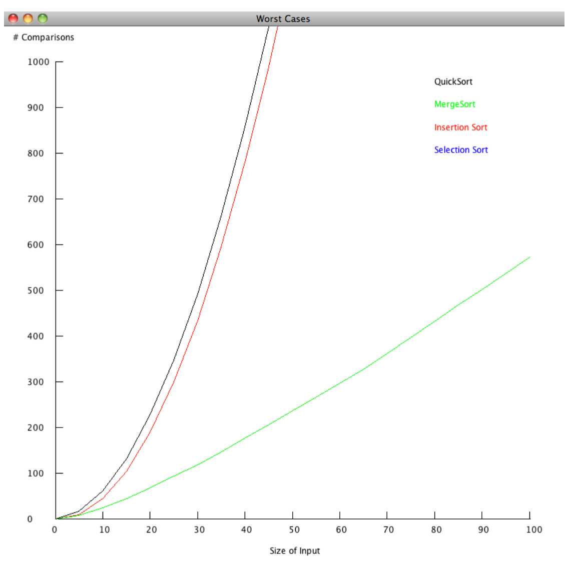
Computer Science



55

Comparing Sorting Algorithms

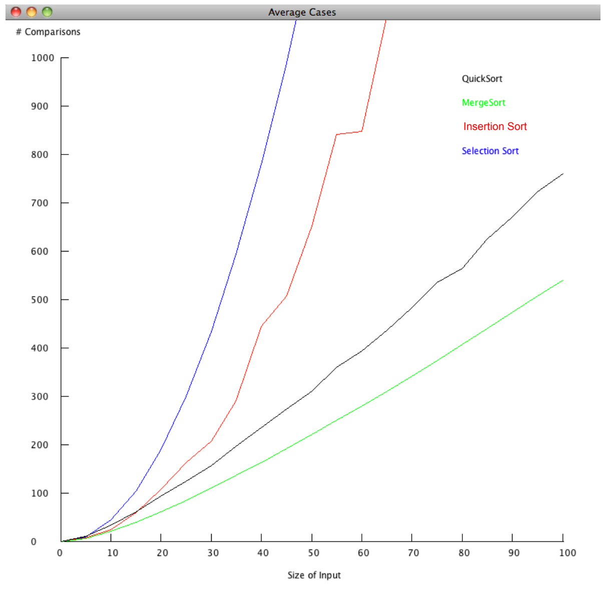
Computer Science



56

Comparing Sorting Algorithms

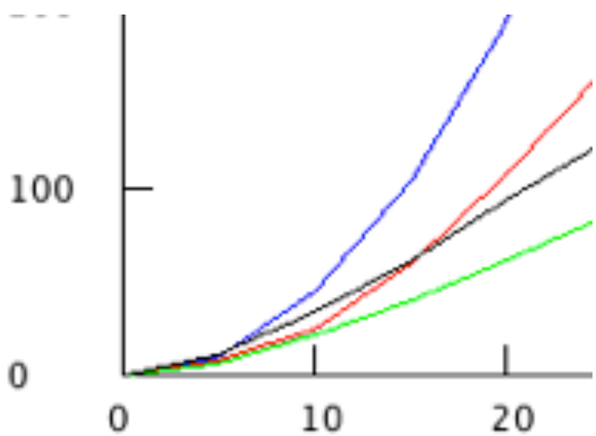
Computer Science



57

Comparison of Sorting Algorithms: Average Case

Computer Science



58

Timing Java Code

Computer Science

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…..

59

Timing Java Code

Computer Science

Here is a simple way to time a region of Java code:

```
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.

60

Timing Java Code

Computer Science

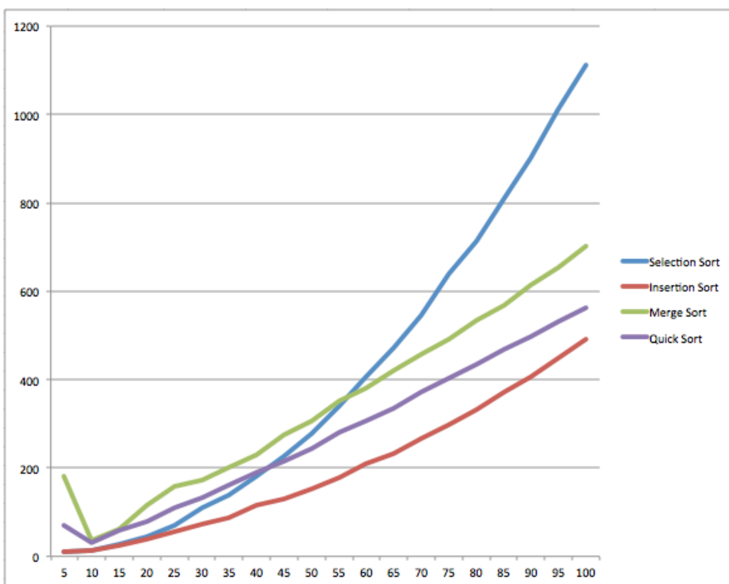
Here is a sample of what I wrote to time our sorting algorithms:

```
for(int i = 5; i <= 200 ; i+=5){
   int[][] a = new int[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));
}
```

61

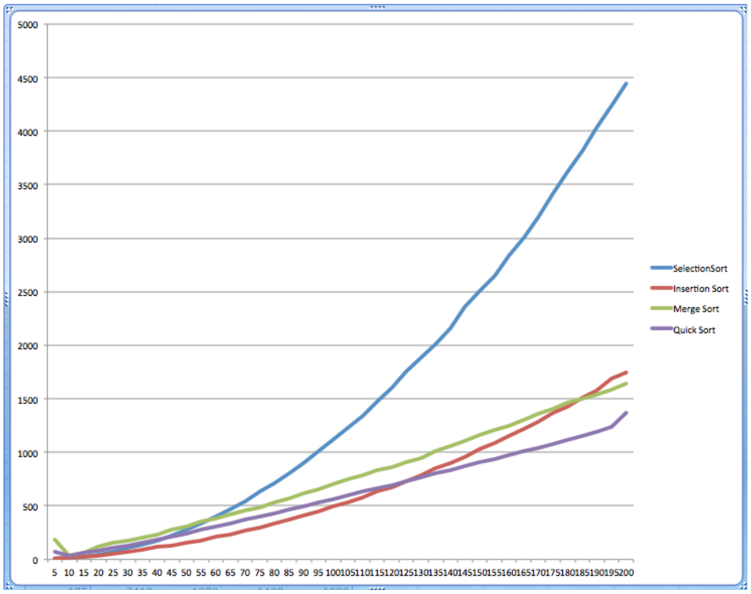
Timing Java Code: Average Case for Actual Time

Computer Science



62

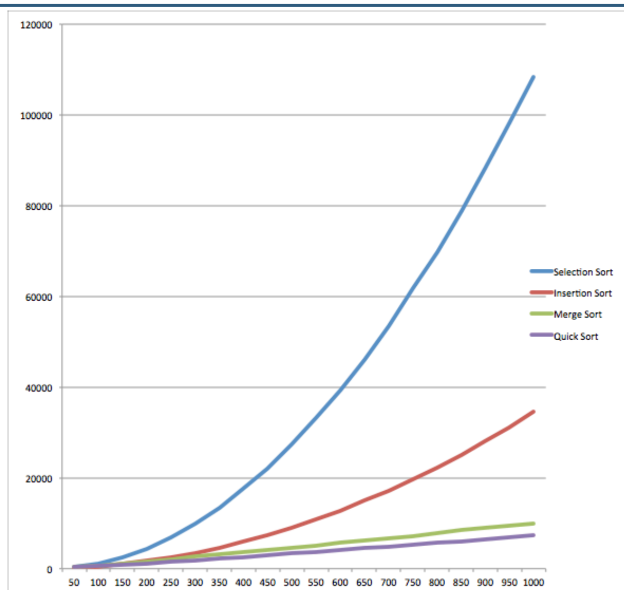
Timing Java Code: Average Case for Actual Time



63

Timing Java Code

BOSTON UNIVERSITY
Computer Science



64

Stability of Sorting Algorithms

A sorting algorithm is called Stable if it keeps identical elements in the same order; suppose we have a list with duplicate 3's, and we color them to distinguish the two different instances of the same element:

1 7 3 5 8 2 3

1 2 3 3 5 7 8

(stable!)

1 2 3 3 5 7 8

(not stable!)

When using a sorting algorithm as a component of another algorithm, such as lexicographic sorting, we may need to know such properties as stability.

Final thoughts on sorting: Stability

Computer Science

Which algorithms we've studied are stable?

Selection Sort ?

Insertion Sort ?

Merge Sort ?

Quick Sort ?

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO!

The problem in general is with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), after the minimal element has been found:

1 2 | 7 5 8 7 3 6 // find the minimum in unsorted part

67

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO!

The problem in general is with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), after the minimal element has been found:

1 2 | 7 5 8 7 3 6 // find the minimum in unsorted part

68

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO!

The problem in general is with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), after the minimal element has been found:

1 2 | 7 5 8 7 3 6 // find the minimum in unsorted part

// swap with top element of unsorted

69

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO!

The problem in general is with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), after the minimal element has been found:

1 2 | 3 5 8 7 7 6 // find the minimum in unsorted part

// swap with top element of unsorted

70

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO!

The problem in general is with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), after the minimal element has been found:

1 2 3 | 5 8 7 7 6

// find the minimum in unsorted part
// swap with top element of unsorted
// continue (but 7's have been swapped!

71

Stable Sorting: Example

Computer Science

Which algorithms we've studied are stable?

Selection Sort ? NO! (can be repaired but why bother?)

Insertion Sort ? Yes, as long as you do not move the element being inserted past duplicates.

Merge Sort ? Yes, as long as when merge compares identical elements it keeps them in the same order.

Quick Sort ? NO!

Quicksort has the same problem as Selection Sort with "long distance swaps," where an element is moved a long distance (possibly over a duplicate), e.g., during the swap in the partition method:

5 2 7 1 3 8 9 3 6 => 5 2 3 1 3 8 9 7 6

72