Divide and Conquer

Divide-and-Conquer

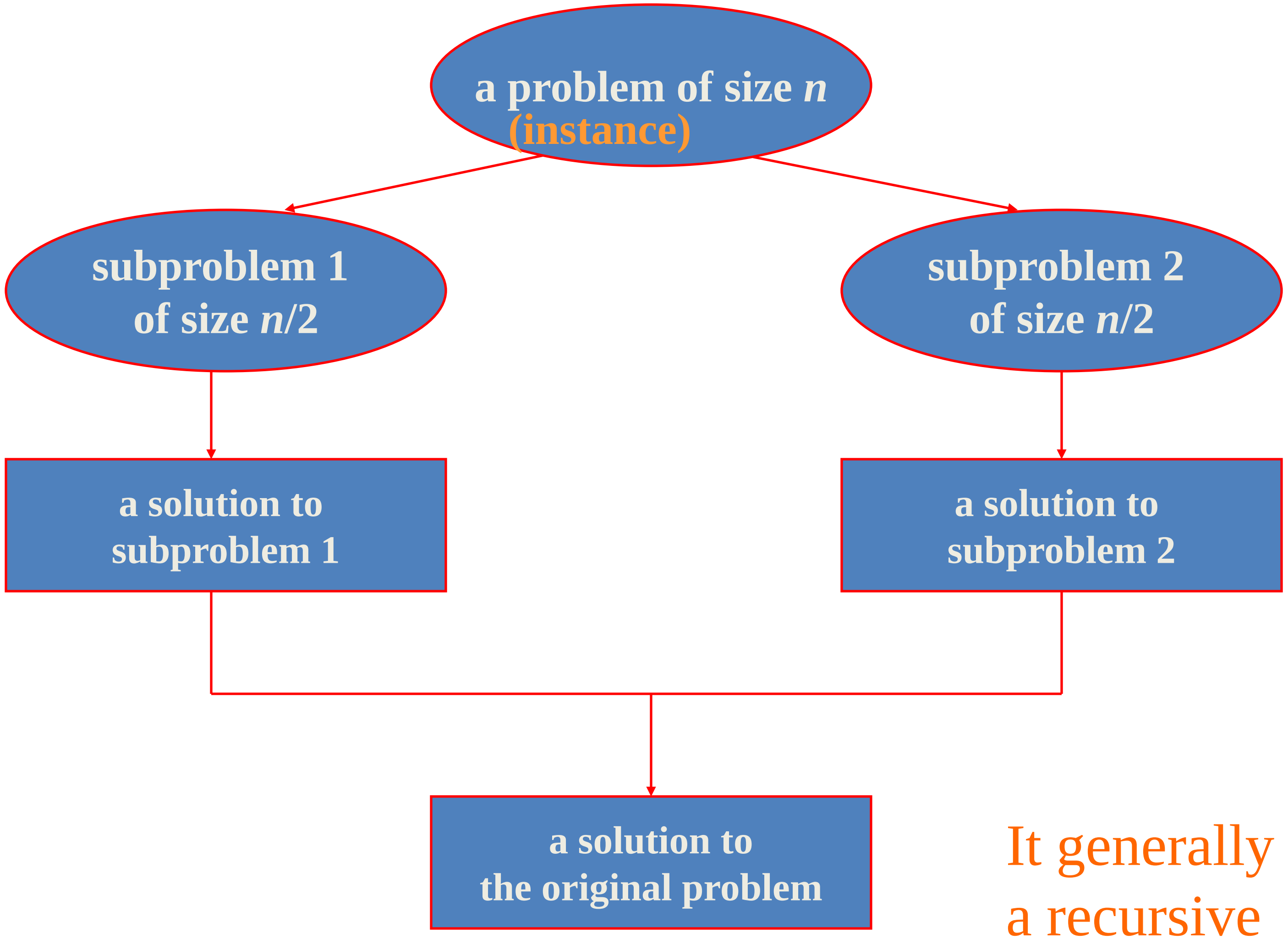
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

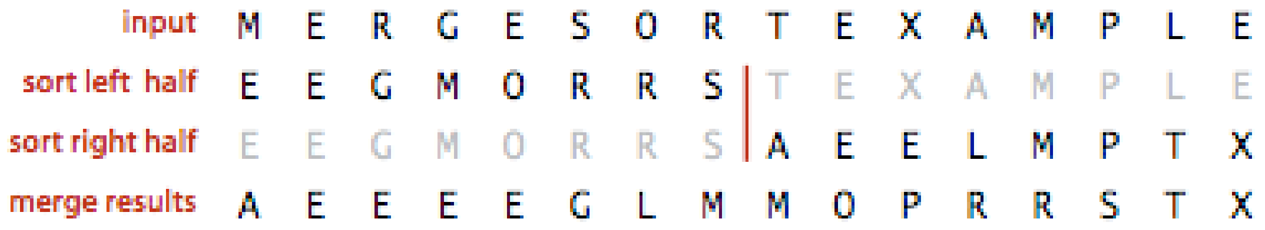
Divide-and-Conquer Technique (cont.)



It generally leads to a recursive algorithm!

Mergesort

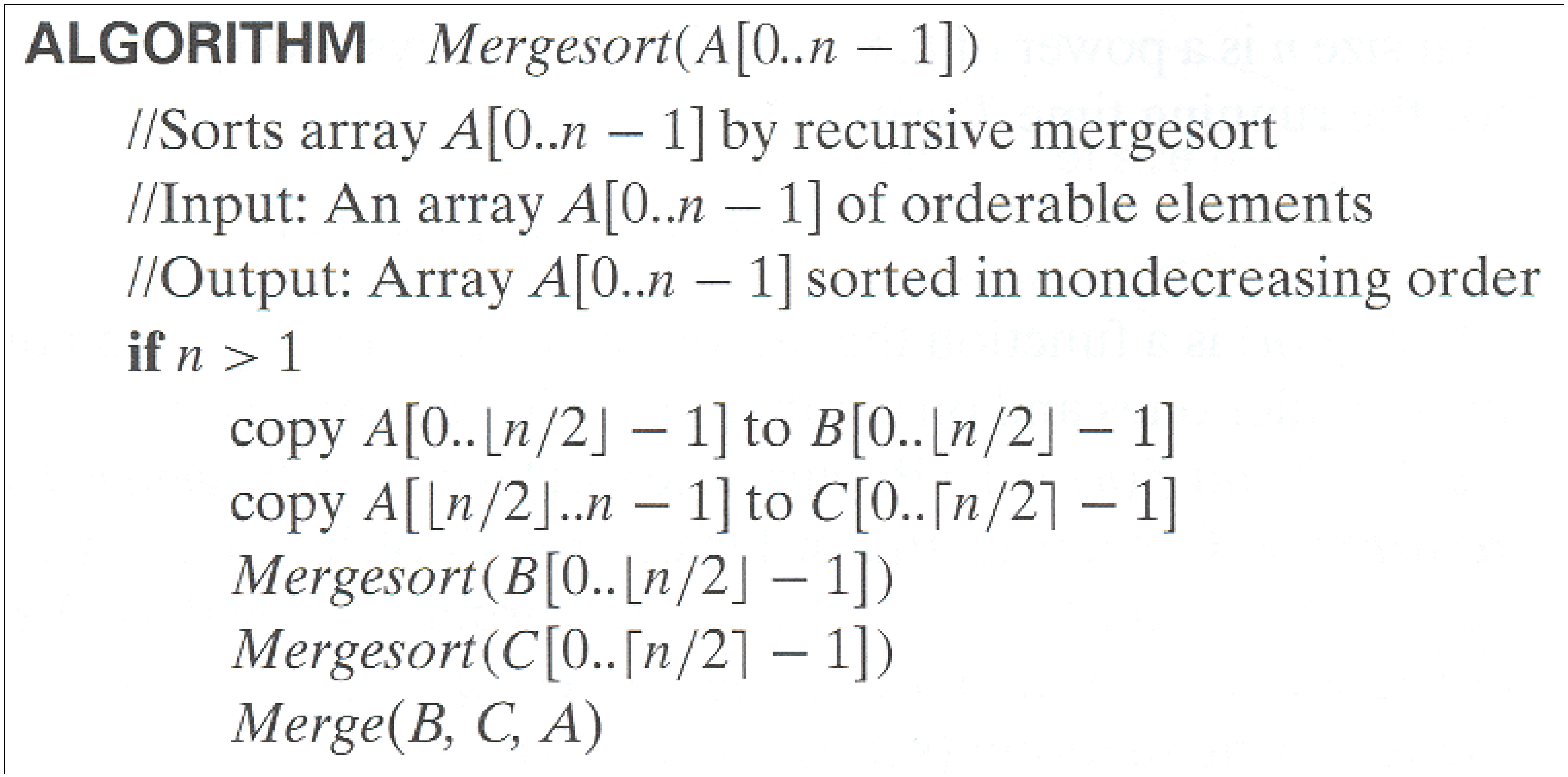
- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  – Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  – Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.



input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Pseudocode of Mergesort



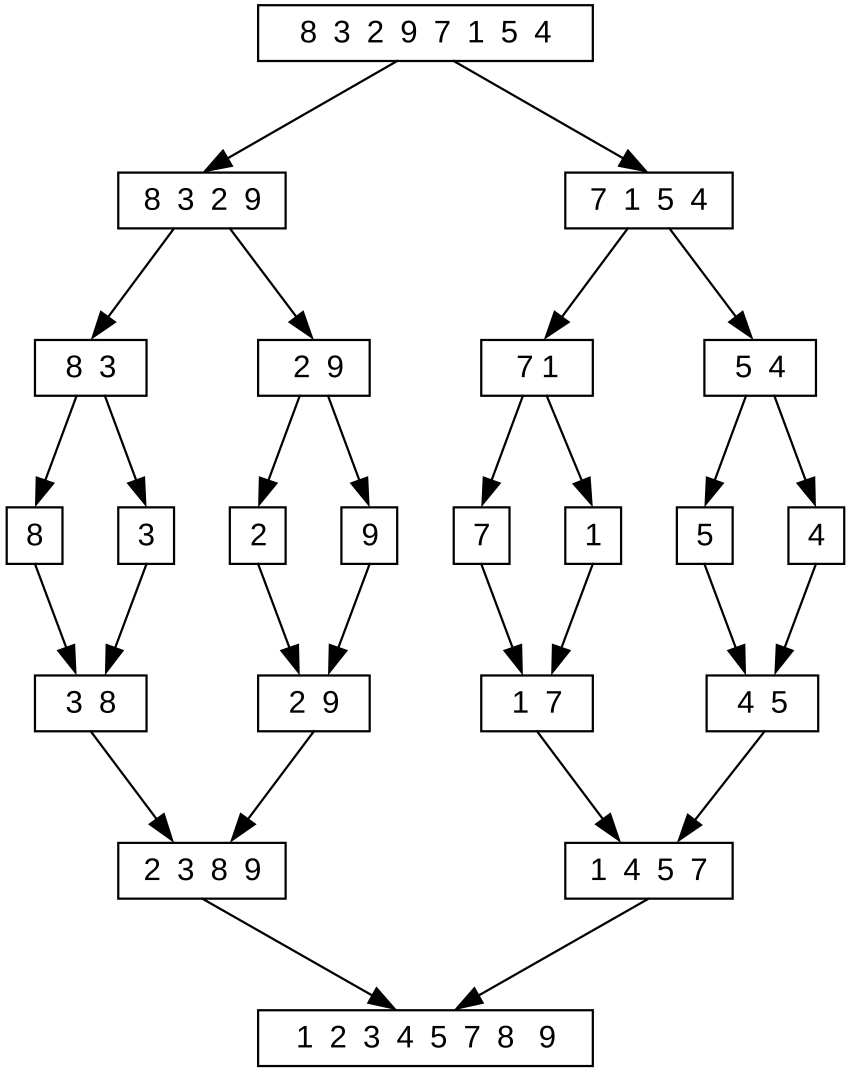
```
ALGORITHM  $Mergesort(A[0..n-1])$
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
        $Mergesort(C[0..\lceil n/2 \rceil - 1])$
        $Merge(B, C, A)$
```

Pseudocode of Merge

```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q - 1]$ to $A[k..p + q - 1]$
    else copy $B[i..p - 1]$ to $A[k..p + q - 1]$
```

Time complexity: **$\Theta(p+q) = \Theta(n)$** comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

See **here**

Analysis of Divide-and-Conquer Recurrence Algorithms

The **master theorem** provides a cookbook solution in asymptotic terms (using Big O notation) for recurrence relations.

We can often represent divide and conquer algorithms as a recurrence releation in the following form:

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Where n is the size of the problem, a is the number of subproblems, n/b is the size of each subproblem, f(n) is the work done outside the recursive calls

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$ **Θ(n^2)**

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$ **Θ(n^2log n)**

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$ **Θ(n^3)**

Note: The same results hold with O instead of Θ.

Analysis of Mergesort

- All cases have same efficiency:

  $$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$$

- For Master Theorem, a = 2, b = 2 , d = 1
  - $a=b^d$ so $T(n) \in \Theta(n \log n)$

- Space requirement: $\Theta(n)$

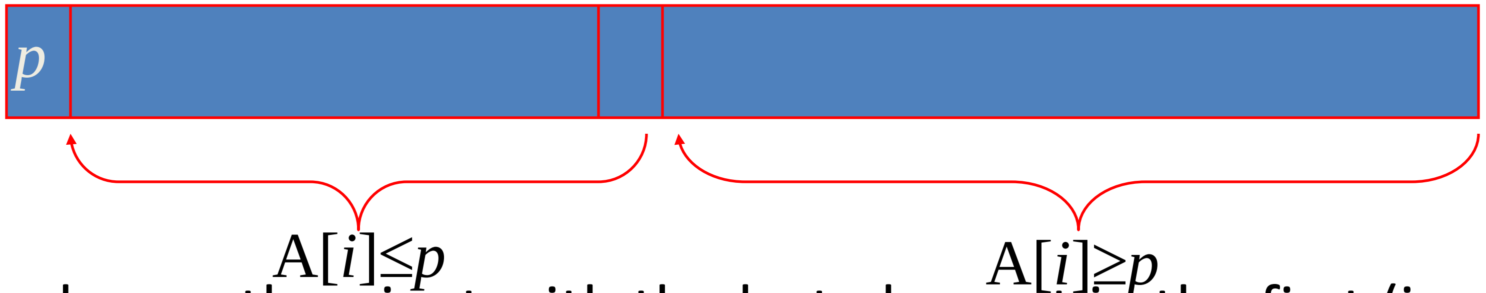
- Can be implemented without recursion (bottom-up)

Improvements

- *Use insertion sort for small subarrays.*
  - you can improve most recursive algorithms by handling small cases differently. Switching to insertion sort for small subarrays will improve the running time of a typical mergesort implementation by 10 to 15 percent.
  - We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quicksort Algorithm-

```
ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[O,n -1], defined by its left and
// right indices l and r
//Output: Subarray A[l .. r] sorted in non-decreasing order
if l < r
s = Partition(A[l .. r]) //s is is a split position
Quicksort(A[l .. s- 1])
Quicksort(A[s + l..r])
```

```
ALGORITHM Partition(A[l .. r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[O .. n - 1], defined by its left and right indices l
// and r (l < r)
//Output: A partition of A[l..r ], with the split position returned as
II this method's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i]>= p
    repeat j ←  j - 1 until A[j] <= p
    swap(A[i], A[j])
until i >=  j
swap(A[i], A[j]) //undo last swap when i >= j
swap(A[l], A[j])
return j
```

Quicksort Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$ **$T(n) = T(n-1) + \Theta(n)$**
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning
    - instead of just taking the first item (or a random item) as pivot, take the median of the first, middle, and last items in the list
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

Very efficient algorithm for searching in <u>sorted array</u>:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

```
$l \leftarrow 0$;   $r \leftarrow n-1$;
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if  $K$ = A[$m$]  return $m$
    else if $K$ < A[$m$]  $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1
```

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

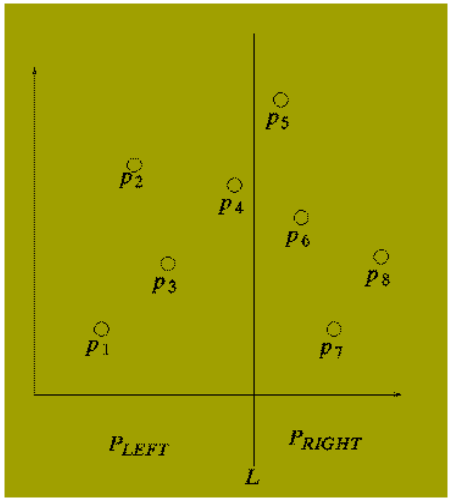
- Bad (degenerate) example of divide-and-conquer
  because only one of the sub-instances is solved

Closest-Pair Problem by Divide-and-Conquer

- Given a set of N points, find the pair with minimum distance
- brute force approach:
    - consider every pair of points, compare distances & take minimum
    - $O(N^2)$

▪ there exists an O(N log N) divide-and-conquer solution

1. sort the points by x-coordinate
2. partition the points into equal parts using a vertical line in the plane
3. recursively determine the closest pair on left side (Ldist) and the closest pair on the right side (Rdist)
4. find closest pair that straddles the line, each within min(Ldist,Rdist) of the line (can be done in O(N))
5. answer = min(Ldist, Rdist, Cdist)



Efficiency of the Closest-Pair Algorithm

Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

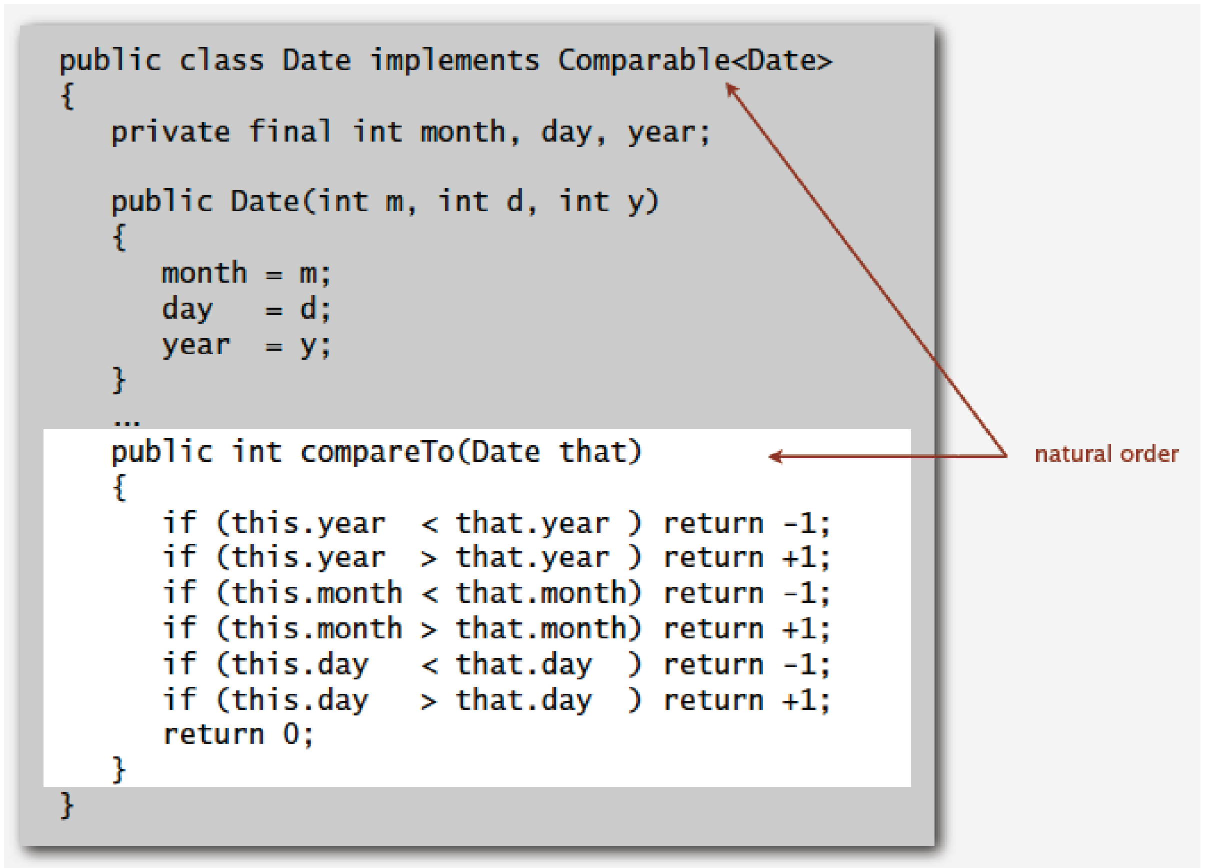
By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in \Theta(n \log n)$$

So the total time is $\Theta(n \log n)$.

Comparable interface: review

- Comparable interface: sort using a type's natural order.



```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }
   ...
   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

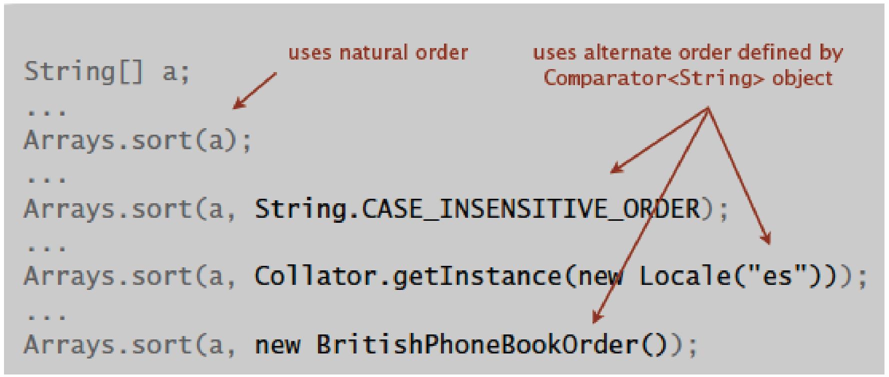
Comparator interface

- Comparator interface: sort using an alternate order.

  Ex. Sort strings by:
- Natural order. Now is the time
- Case insensitive. is Now the time
- Phone book. McKinley Mackintosh
- � . . .

Comparator interface: system sort

- To use with Java system sort:
- Create Comparator object.
- Pass as second argument to Arrays.sort().


uses natural order
uses alternate order defined by Comparator<String> object

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
```

- Can decouple the definition of the data type from the definition of what it means to compare two objects of that type.

Comparator interface: implementing

- To implement a comparator:
  - Define an inner class(next topic) that implements the Comparator interface.
  - Implement the compare() method.

```java
public class Student
{
   public static final Comparator<Student> BY_NAME    = new ByName();
   public static final Comparator<Student> BY_SECTION = new BySection();
   private final String name;
   private final int section;
   ...

   private static class ByName implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }

   private static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

one Comparator for the class

this technique works here since no danger of overflow

`Arrays.sort(a, Student.BY_NAME);`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

`Arrays.sort(a, Student.BY_SECTION);`

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

References

http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

http://algs4.cs.princeton.edu/home/