

## Divide-and-Conquer: Multiplication

- $981 * 1234$

		shift	
09	12	4	108....
09	34	2	306..
81	12	2	972..
81	34	0	2754
			<hr/>
			1210554

- runtime:  $O(n^2)$ 
  - Can be improved!
  - need to reduce number of multiplications

© 2005-06. 324-1-72

## Divide-and-Conquer Multiplication

- $981 = 100w + x$
- $1234 = 100y + z$

$$\begin{aligned} 981 * 1234 &= (100w+x) * (100y + z) \\ &= 10,000wy + 100(wz+xy) + xz \end{aligned}$$

$$p = wy$$

$$q = xz$$

$$r = (w+x)*(y+z) = wy + wz + xy + xz$$

$$981*1234 = 10,000p + 100(r-p-q) + q$$

- Now it's only 3 multiplications in each step!
- Runtime:  $O(n^{\lg 3})$ 
  - $\lg 3 \approx 1.59$

© 2005-06. 324-1-73

## Divide-and-Conquer: Template

```
Object divideAndConquer( Set x )
  if( x.size() < threshold ) {
    return baseAlgorithm( x );
  }
  Set< Set > subsetsOfX = decompose( x );
  Map< Set, Object > solution = new Map<>< Set, Object >;
  for( Set s: subsetsOfX ) {
    solution.put( s, divideAndConquer( s ) );
  }
  return recombine( solution );
}
```

© 2005-06. 324-1-74

## Runtime for Divide-and-Conquer

- $l$ : number of subinstances
- $n/b$ : size of a subinstance
- $g(n)$ : runtime of everything except the call to itself
- $k$ :  $g(n) \in \Theta(n^k)$
  
- Runtime:  $t(n) = l \cdot t(n/b) + g(n)$
  
- Solution of the recurrence:
  - $\Theta(n^k)$  if  $l < b^k$
  - $\Theta(n^k \log n)$  if  $l = b^k$
  - $\Theta(n^{\log_b l})$  if  $l > b^k$
  
- Example: Multiplication of large integers
  - $l = 3, b = 2, k = 1$
  - runtime:  $O(n^{\lg 3})$

© 2005-06. 324-1-75

## Binary Search

- Searching in an array
  - if unsorted: check each element
  - $O(n)$
- If sorted?

```
int search( Comparable[] t, Comparable x ) {
    if( x < t[0] || x > t[ t.length - 1 ] ) {
        return -1;                // not found
    } else {
        int result = searchRecursive( t, x, 0, t.length - 1 );
        if( ! t[ result ] == x ) {
            return -1;            // not found
        }
        return result;
    }
}
```

© 2005-06. 324-1-76

## Binary Search

```
int searchRecursive(Comparable[] t, Comparable x, int left, int right ) {
    if( left == right ) {
        return left;
    }
    int middle = ( left + right ) / 2;
    if( x <= t[ middle ] ) {
        return searchRecursive( t, x, left, middle )
    } else {
        return searchRecursive( t, x, middle+1, right );
    }
}
```

- Runtime?

© 2005-06. 324-1-77

## Sorting through Divide-and-Conquer

- Ideas??

© 2005-06. 324-1-78

## Sorting

- Mergesort
  - split array into halves
  - sort each half
  - merge to halves (preserving order)
- Quicksort
  - split array around a “pivot”
  - put all elements that are smaller than pivot before pivot
  - put all elements that are bigger than pivot after pivot
  - sort each part

© 2005-06. 324-1-79

## Mergesort

```
void mergesort( Comparable[] target ) {
    mergesortRec( target, 0, target. length -1,
                 new Comparable[ target. length ] )
}

void mergesortRec(Comparable[] target, int left, int right,
                  Comparable [] aux ) {
    if( right <= left ) {
        return;
    }
    int middle = ( right + left ) / 2;
    mergesortRec( target, left, middle );
    mergesortRec( target, middle + 1, right );
    merge( target, left, middle, right, aux ); //merge two halves
}
```

© 2005-06. 324-1-80

## Merging two parts of an Array

```
void merge(Comparable[] target, int left, int middle, int right,
           Comparable[] aux ) {
    int i, j;
    for( i = left; i <= middle; i++ ) {
        aux[ i ] = target[ i ] ;
    }
    for( j = middle; j < right; j++ ) {
        aux[ right + middle - j ] = target[ j ];
    }
    for( int k = left; k <= right; k ++ ) {
        if( aux[ j ] < aux[ i ] ) {
            target[ k ] = aux[ j-- ];
        } else {
            target[ k ] = aux[ i++ ];
        }
    }
}
```

- puts the array in bitonic order: up, then down
- alternative: use sentinels (infinitely large values at end of each part)

© 2005-06. 324-1-81

## Mergesort

- Divide-and-conquer algorithm for sorting
- Runtime:
  - $l = 2, b=2, k=1$
  - $O(n \log n)$
- Space requirements:
  - $2n$
- What happens if one does not split array in middle?
  - e.g. split:  $(n-1,1)$
  - runtime:  $O(n^2)$  !!

© 2005-06. 324-1-82

## Quicksort

- Another divide-and-conquer sort algorithm!
- Instead of merging subinstances...
  - we compute the subinstances in a smart way

```
void quicksort( Comparable[] target, int left, int right ) {
    if( right <= left ) {
        return;
    }
    int i = partition( target, left, right );
    quicksort( target, left, i - 1 );
    quicksort( target, i + 1, right );
}
```

© 2005-06. 324-1-83

## Partition (or Pivot)

```
int partition( Comparable[] target, int left, int right ) {
    Comparable pivot = target[ left ] ;
    int k = left;
    int result = right + 1;
    repeat { k++; } until( target[ k ] > pivot || k>= right );
    repeat { result--; } until( target[ result ] <= pivot );
    while( k < result ) {
        swap( target, k, result );
        repeat{ k++ } until ( target[ k ] > pivot );
        repeat{ result-- } until( target[ result ] <= pivot );
    }
    swap( target, left, result ); //put pivot in its place
    return result;
}
```

- Puts all smaller elements before result, and all larger elements after it

© 2005-06. 324-1-84

## Quicksort

- Worst-case:
  - each pivot element is at an extreme
  - results in unbalanced splits
  - $O(n^2)$
- Average case:
  - if elements are randomly distributed
  - $O(n \log n)$
- Can be sped up by using a different base algorithm
  - for example, selection sort
- On average, fastest sorting algorithm!
  - hidden constants smaller than heap sort, mergesort
- No extra space required

© 2005-06. 324-1-85

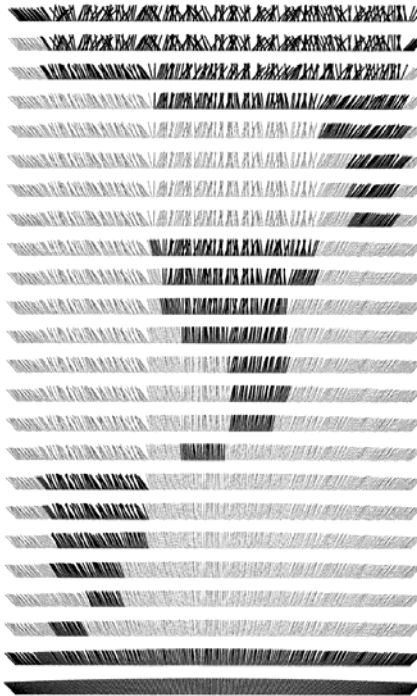
## Overview of Sorting Algorithms

insertion sort	$O(n^2)$	
selection sort	$O(n^2)$	
heapsort	$O(n \log n)$	
mergesort	$O(n \log n)$	extra space needed
quicksort	$O(n^2)$	$O(n \log n)$ on average; fastest on average

© 2005-06. 324-1-86

**Figure 7.9**  
Comparisons in quicksort

Quicksort subfiles are processed independently. This picture shows the result of partitioning each subfile during a sort of 200 elements with a cutoff for files of size 15 or less. We can get a rough idea of the total number of comparisons by counting the number of marked elements by column vertically. In this case, each array position is involved in only six or seven subfiles during the sort.



Source: Sedgewick,  
Algorithms, 3d ed., 2003  
© 2005-06. 324-1-87