

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. 324-1-69

Divide-and-Conquer Multiplication

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

$$981 * 1234 = (100w+x) * (100y + z)$$

$$= 10,000wy + 100(wz+xy) + xz$$

$$p = wy$$

$$q = xz$$

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

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

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

© 2005. 324-1-70

Divide-and-Conquer: Template

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

© 2005. 324-1-71

Runtime in General

- l : number of subinstances
- n/b : size of a subinstance
- $g(n)$: runtime of everything except the call to itself
- k : $g(n) \in O(n^k)$

- Runtime: $t(n) = l \cdot t(n/b) + g(n)$

- Solution of the recurrence:
 - $O(n^k)$ if $l < b^k$
 - $O(n^k \log n)$ if $l = b^k$
 - $O(n^{\log_b l})$ if $l > b^k$

- Example: Multiplication of large integers
 - $l = 3, b = 2, k = 1,$
 - runtime: $O(n \lg^3)$

© 2005. 324-1-72

Binary Search

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

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

© 2005. 324-1-73

Binary Search

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

- Runtime?

© 2005. 324-1-74

Sorting through Divide-and-Conquer

- Ideas??

© 2005. 324-1-75

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. 324-1-76

Mergesort

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

void mergesortRec( Object[] target, int left, int right, Object[] 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. 324-1-77

Merging two parts of an Array

```
void merge( Object[] target, int left, int middle, int right, Object[] aux ) {
    int i, j;
    for( i = middle+1; i > left; i-- ) {
        aux[ i - 1 ] = target[ i - 1 ];
    }
    for( j = middle; j < right; j++ ) {
        aux[ right + middle - j ] = target[ j + 1 ];
    }
    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. 324-1-78

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. 324-1-79

Quicksort

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

```
void quicksort( Object[] 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. 324-1-80

Partition (or Pivot)

```
int partition( Object[] target, int left, int right ) {
    Object 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], target[result]);
        repeat{ k++; } until ( target[ k ] > pivot );
        repeat{ result-- } until( target[ result ] <= pivot );
    }
    swap( target[ left ], target[ result ] );
    return result;
}
```

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

© 2005. 324-1-81

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

© 2005. 324-1-82

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

© 2005. 324-1-83