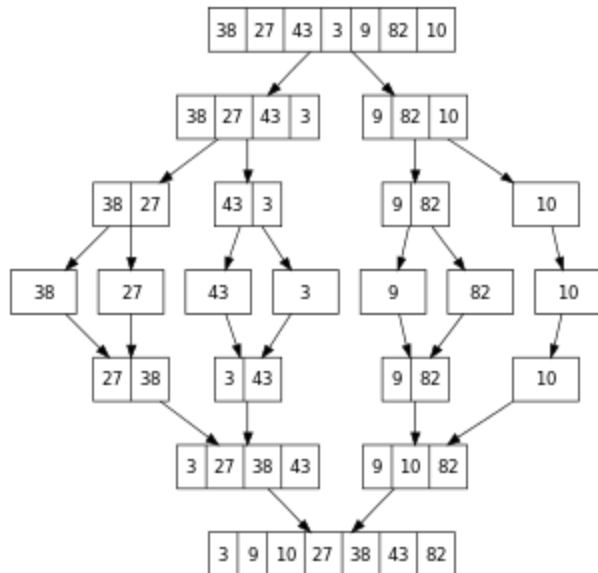


Chapter 4

Divide and Conquer



References:

[KT 5.1, 5.5]

[DPV 2.1, 2.3, 2.4]

[CLRS 4.1, 7, 9]

[SSS 4.5, 4.6, 4.10]



Algorithm Design Technique #1

- Principle of *Divide and Conquer*:
 - Partition a problem into different parts
 - Solve the subproblems
 - Combine the solutions to form the overall solution
- Very often, we solve a smaller version of the same problem
 - Use recursion!
 - We don't need to care how the smaller subproblems are solved
 - In fact, they will be reduced to even smaller subproblems, and so on...
 - Stop at base case (small problem size)

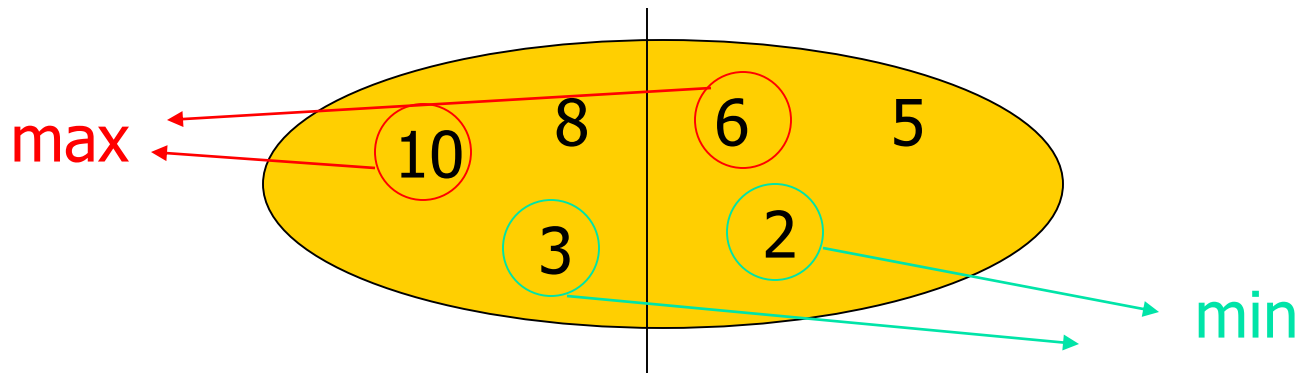


Finding Max and Min

- Problem: given n numbers, *find their maximum and minimum*
- Obvious solution: find separately
 - find maximum using $n - 1$ comparisons
 - find minimum using $n - 1$ comparisons
 - total $2n - 2$ comparisons
- Can we do better?
 - $2n - 3$ is enough (how?)
 - Still better?

Find Max and Min: Idea

- [Divide] Divide the n numbers into two halves, $S1$ and $S2$
- [Solve] For each half, find the max and min
 - How? Recursively!
- [Combine] $\max(S) = \max(\max(S1), \max(S2))$,
 $\min(S) = \min(\min(S1), \min(S2))$





MaxMin: D&C Algorithm

- The algorithm:

```
MaxMin(S[1..n])
// return max and min of elements in S

if (n == 1) {           // base case, size=1
    max := S[1]; min := S[1];
}
else if (n == 2) { // base case, size=2
    if (S[1]>S[2]) {max := S[1]; min := S[2];}
    else          {max := S[2]; min := S[1];}
}
else {
    (... continue on next page)
```



MaxMin: D&C Algorithm (cont'd)

(... follows from previous page)

Divide S into S1 and S2, each with half
of the elements

```
(max1, min1) := MaxMin(S1)  // recursion
```

```
(max2, min2) := MaxMin(S2)
```

```
if (max1 > max2) max := max1
```

```
else                max := max2
```

```
if (min1 < min2) min := min1
```

```
else                min := min2
```

```
}
```

```
return (max,min)
```

- How many comparisons?



Analysis of MaxMin

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

$$\begin{aligned} T(n) &= 2 T(n/2) + 2 \\ &= 2 (2 T(n/4) + 2) + 2 \\ &= 2^2 T(n/2^2) + 2^2 + 2 \\ &= 2^k T(n/2^k) + (2^k + 2^{k-1} + \dots + 2) \\ &= (n/2)T(2) + 2(n/2 - 1) \\ &= 3n/2 - 2 \end{aligned}$$

- Better than $2n - 3$!
- [There is another way of achieving the same number of comparisons. Can you figure it out?]



Sorting Revisited

- Recall: we know a number of sorting algorithms, all in $O(n^2)$ time
 - Selection sort
 - Insertion sort
- Are there faster algorithms?
- Divide and Conquer?
 - Divide the n numbers into two halves
 - Sort the two halves recursively
 - Then?



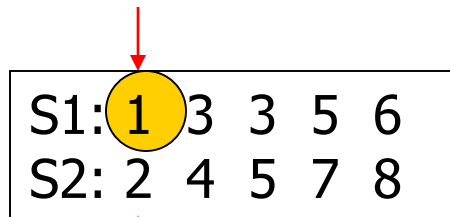
Merge Sort

- Combine the two sorted halves into one
 - How? [imagine: merge two piles of sorted paper]
- Assuming we know how to do it, then we have a new sorting algorithm!

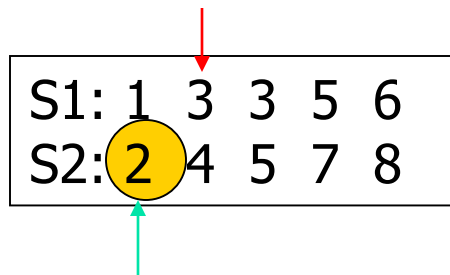
```
MergeSort(A[1..n]) {  
    if (size of A == 1) return A;  
        // base case, nothing to sort  
    B1 := MergeSort(A[1..n/2]);  
    B2 := MergeSort(A[n/2+1..n]);  
    B := Merge(B1, B2);  
    return B; // sorted array  
}
```

Merging Two Sorted Lists

- Given two sorted lists S1 and S2, merge them into a single sorted list
- Idea: maintain an index (pointer) to each list
 - Example:



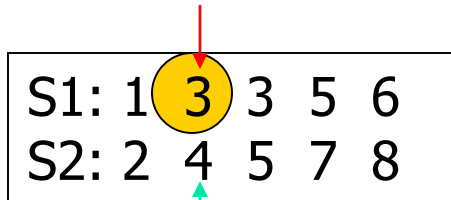
1



1 2

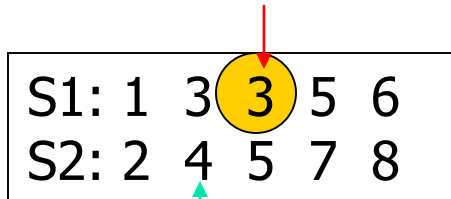
Merging (cont'd)

S1: 1 3 3 5 6
S2: 2 4 5 7 8



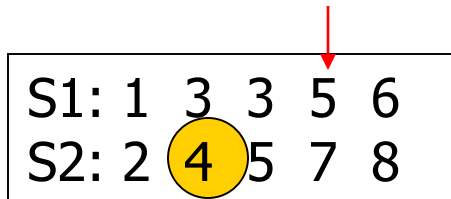
1 2 3

S1: 1 3 3 5 6
S2: 2 4 5 7 8



1 2 3 3

S1: 1 3 3 5 6
S2: 2 4 5 7 8



1 2 3 3 4

Merging (cont'd)

S1: 1 3 3 5 6
S2: 2 4 5 7 8

1 2 3 3 4 5

S1: 1 3 3 5 6
S2: 2 4 5 7 8

1 2 3 3 4 5 5

S1: 1 3 3 5 6
S2: 2 4 5 7 8

1 2 3 3 4 5 5 6

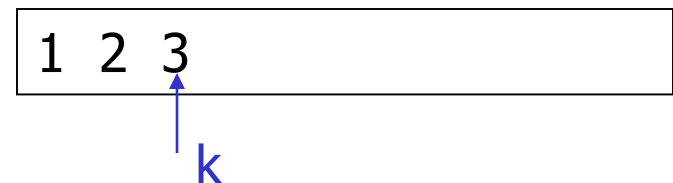
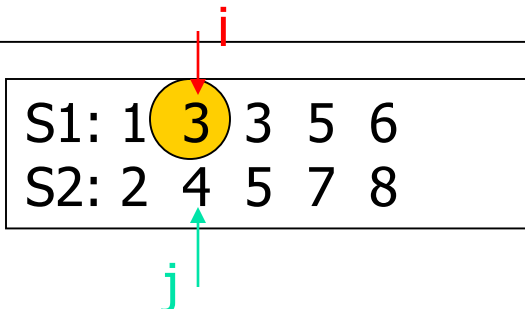
S1: 1 3 3 5 6
S2: 2 4 5 7 8

1 2 3 3 4 5 5 6 7 8

End of S1 reached, just copy 7 and 8

Merge: in Pseudocode

```
/* merge two sorted array A[1..n] and B[1..n]
   into output C[] */
Merge() {
    i := 1; j := 1; k := 1;
    // pointer to arrays A, B, C
    while (i <= n and j <= n) {
        if (A[i] < B[j])
            {C[k] := A[i]; i++; k++; }
        else {C[k] := B[j]; j++; k++; }
    }
    Copy all remaining elements;
}
```





Complexity of Merge

- Code inside while loop: $O(1)$ time
- How many times has the while loop executed?
 - Observation: every time the while loop is executed, *one of i or j is increased by one*
 - i or j can be increased at most n times each
 - So, while loop executed $O(n)$ times at most
- Code after while loop: $O(n)$
- So, total complexity = $O(n)$



Complexity of MergeSort

- Recurrence:
 - $T(n) = 2 T(n/2) + O(n)$
 - So $T(n) = O(n \log n)$ [Case 2 of Master Theorem]
- What is bad about MergeSort?
 - Memory: use another array
 - Involve copying of elements (not in-place sorting)

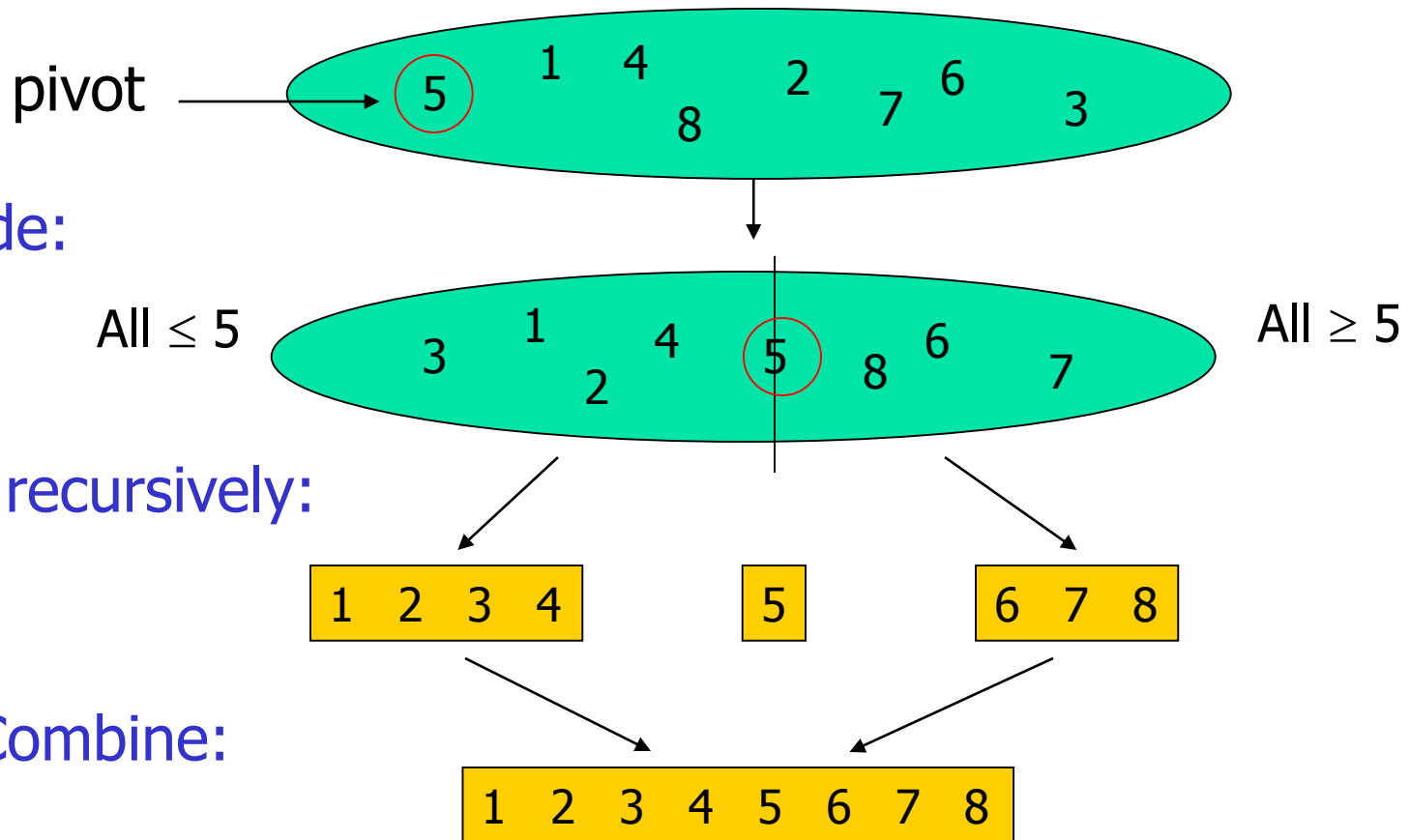


Another Sorting Algorithm: QuickSort

- In MergeSort, merge takes $O(n)$ time
 - Can we avoid the merge?
- Another possible way of doing divide and conquer:
 - Divide the numbers in two sets such that “small” numbers in one set, “large” numbers in another
 - Sort recursively
 - Combine: just put results together! (no merging)

Idea of QuickSort

- In diagrams:





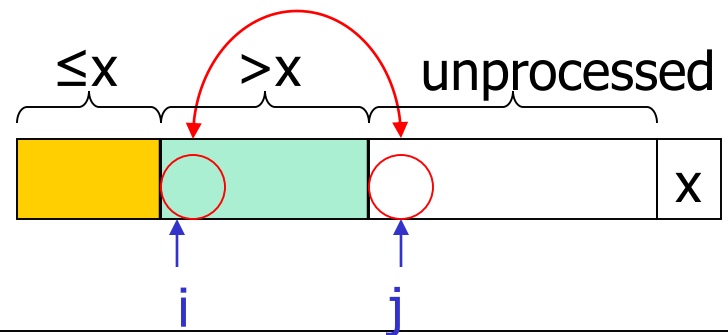
Partition

- Key step: *partitioning the array A*
 - Goal: all elements in left side \leq all elements in right side
 - Choose a “pivot” v (any element in A)
 - A simple pivot: just the last element
 - Move all elements $\leq v$ to the left side of the array
 - Move all elements $> v$ to the right side of the array
 - Return the partition point
- How to do this efficiently?
- Time complexity?

Partition: the Algorithm

- In pseudocode:

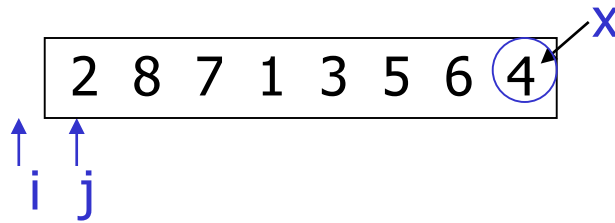
```
Partition(A, p, r) { // partition A[p..r]
  x := A[r] // pivot
  i := p-1
  for j := p to r-1 {
    if (A[j] <= x) {
      i++
      swap(A[i], A[j])
    }
  }
  swap(A[i+1], A[r])
  return i+1
}
```



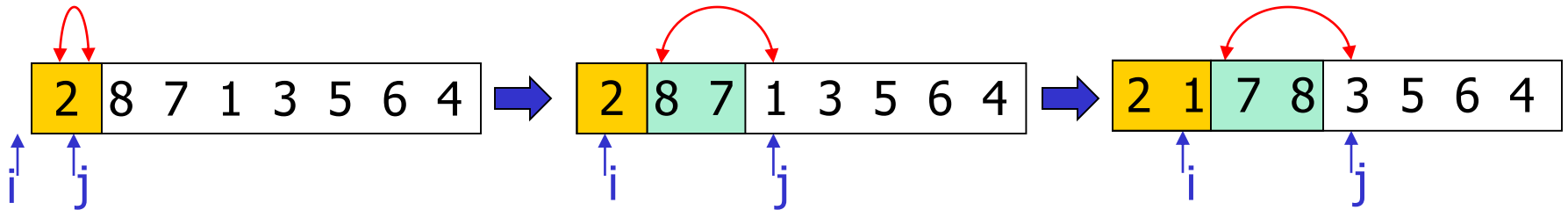
- Running time: $O(n)$

Partition: in Figures

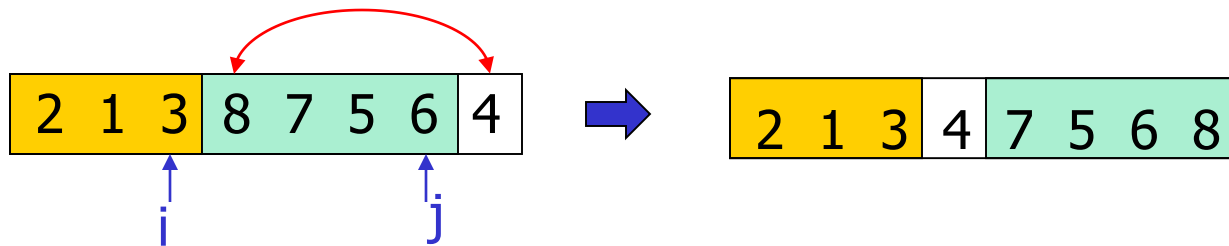
Initially:



Immediately before each time "if" returns true:



For loop finishes:





QuickSort

- With Partition, we can now very easily sort the array recursively
- Pseudocode:
 - Initial call: QuickSort(A, 1, n)

```
QuickSort(A, p, r) {  
    // sort the array A[p..r]  
    if (p < r) {  
        q := Partition(A, p, r)  
        QuickSort(A, p, q-1)  
        QuickSort(A, q+1, r)  
    }  
}
```



Analysis of QuickSort

- Depends on how “well” we partition the list
- Assume the two parts $A[p..q-1]$ and $A[q+1..r]$ have roughly equal size
 - $T(n) = 2 T(n/2) + O(n)$
 - This gives $T(n) = O(n \log n)$
- However, in the worst case:
 - The pivot is the smallest/largest element, so one side of the partition contains $n - 1$ elements
 - $T(n) = T(n-1) + O(n)$
 - This gives $T(n) = c(n + (n-1) + \dots + 1) = O(n^2)$
 - Same as other slow sorting algorithms
 - What is the worst-case input?



Average and Randomized Case

- Average case:
 - Assume all permutations of inputs are equally likely
 - Can be shown to be $O(n \log n)$
- *Randomized algorithms*: uses some random inputs in the algorithm
 - Non-randomized algorithms are called *deterministic*
 - We can randomly choose an element in the array as pivot
 - It can be shown that the expected (average) running time of QuickSort using a random pivot is $O(n \log n)$
- In practice, QuickSort is the fastest sorting algorithm
 - E.g. Java Arrays.sort()



MergeSort vs. QuickSort

- Similar:
 - Both use divide and conquer
 - Divide/combine work $O(n)$ time
- Different:

Mergesort	Quicksort
Two subproblems always have equal size	Two subproblems may not have equal size
Divide is trivial (just any way), combine is difficult (merge sorted lists)	Divide is difficult (find good split), combine is trivial (just put them together)
Worst-case $O(n \log n)$; not good in practice	Worst-case $O(n^2)$, but average case $O(n \log n)$ and works well in practice

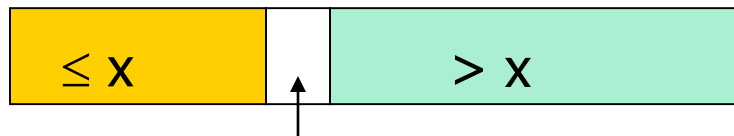


Order Statistics

- Can guarantee good partitioning if choose the *median* as pivot
- Median: the $\lceil n/2 \rceil$ -th number
 - The “middle number” according to increasing order
- More generally: the *k-th order statistic* is the k-th smallest number
 - E.g. minimum = 1st order statistic
 - median = $\lceil n/2 \rceil$ -th order statistic
- How to find the k-th order statistic (for some given k)?
 - For $k = 1$ or n , we know how to do in $O(n)$ time (how?)
 - $k = 2$?

Finding Order Statistic

- Some attempts:
 - Obvious solution: sort the n numbers. $O(n \log n)$ time
 - Another solution: find the min, throw it away, find the min from the rest, throw it away... $O(nk)$ time
 - Anything better? Divide and conquer?
- Reduce to smaller problem:
 - Recall: Partition returns the index of the “partition point”
 - Observation: once we know this index, we know which part of the array the k -th order statistic is in



7-th element

where is 10-th order statistic?



Using Partition to Help Select

- Formally, let $q = \text{Partition}(A, 1, n)$
 - If $k = q$, gotcha
 - If $k < q$, find k -th order statistic in $A[1..q-1]$
 - If $k > q$, find $(k - q)$ -th order statistic in $A[q+1..n]$
- The algorithm:

```
Select(A, p, r, k) {  
    // return k-th smallest number in A[p..r]  
    if (p==r) return A[p] // base case  
    q := Partition(A,p,r)  
    len := q - p + 1  
    if (k == len)    return A[q]  
    else if (k<len)  return Select(A,p,q-1,k)  
    else             return Select(A,q+1,r,k-len)  
}
```



Time Complexity of Select()

- If Partition() gives good split:
 - $T(n) = T(n/2) + O(n)$
 - This gives $T(n) = O(n)$
- However, Partition may be unbalanced:
 - $T(n) = T(n - 1) + O(n)$
 - This gives $T(n) = O(n^2)$
- If we use randomized algorithms (choose a random pivot), it can be shown that the expected running time is $O(n)$



Improving the Worst-case Time

- “On average” it is likely to be $O(n)$
 - Example: even if it is always a 10/90 split
 - $T(n) = T(9n/10) + O(n)$
 - Answer is still $T(n) = O(n)$
- Can we find order statistic in worst-case linear-time?
 - Yes but a slightly different algorithm (omitted)
- Returning to Quicksort...
 - We can use the worst-case $O(n)$ time algorithm to find the median in `Partition()`
 - Using the median as pivot in `Partition()`, gives a worst-case $O(n \log n)$ time sorting algorithm



Integer Multiplication



Bitwise Operations for Addition

- What is the time complexity of basic arithmetic operations?
 - $O(1)$ time if they fit into machine word size (e.g. 32 bits)
 - Larger than that?
- A slightly different model: assume *bit operations* take $O(1)$ time
- How to add two n -bit numbers?
 - Straightforward method: $O(n)$ time

$$\begin{array}{r} 24175 \\ + 3489 \\ \hline 27664 \end{array}$$

(For simplicity, we use digits instead of bits in examples)

- $O(n^2)$

At most $2n$ columns
Summing each column takes
 $O(n)$ time

Multiplication Using D&C

- Try divide and conquer?
- Consider 2-bit x 2-bit:
 - $(2a+b) \times (2c+d)$
 - $= 2^2(ac) + 2(ad) + 2(bc) + (bd)$
 - Four multiplications of 1-bit x 1-bit number
- n-bit numbers:

$$\underbrace{\boxed{a}}_{n/2 \text{ bits}} \underbrace{\boxed{b}}_{n/2 \text{ bits}} = 2^{n/2} \times \boxed{a} + \boxed{b}$$

- Example:

$$\boxed{123} \boxed{456} = 10^3 \times \boxed{123} + \boxed{456}$$



Multiplication Using D&C (cont'd)

- In general, for two n -bit numbers:
 - $(2^{n/2} a + b) \times (2^{n/2} c + d)$
 - $= 2^n (ac) + 2^{n/2} (ad + bc) + bd$
 - Each of ac , ad , bc , bd are $(n/2) \times (n/2)$ -bit multiplications
- Time complexity:
 - Four subproblems of size $n/2$
 - Some additions (at most $2n$ bits), $O(n)$ time
 - 2^n are just bit shifts, also $O(n)$ time
 - $T(n) = 4 T(n/2) + O(n)$
 - Solution: $T(n) = O(n^2)$. No improvement!



Karatsuba Algorithm

- However, observe that
 - $ad + bc = (a+b)(c+d) - ac - bd$
 - We already have ac, bd
 - Just one more multiplication of $(a+b)(c+d)$ will do!
 - Saved one multiplication at the expense of more additions/subtractions (take $O(n)$ time only)
 - The save happens at each level of recursion
- Time complexity:
 - Three subproblems of size $n/2$, addition/shifts still take $O(n)$ time
 - $T(n) = 3 T(n/2) + O(n)$
 - Solution: $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$