# CS 4102, Algorithms: More Divide and Conquer

- Read: *Algorithms* text, Chapter 5
- Examples:
  - Mergesort
  - Trominos
  - Closest Pair of Points
  - Strassen’s Matrix Multiplication Algorithm

# New Problem: Sorting a Sequence

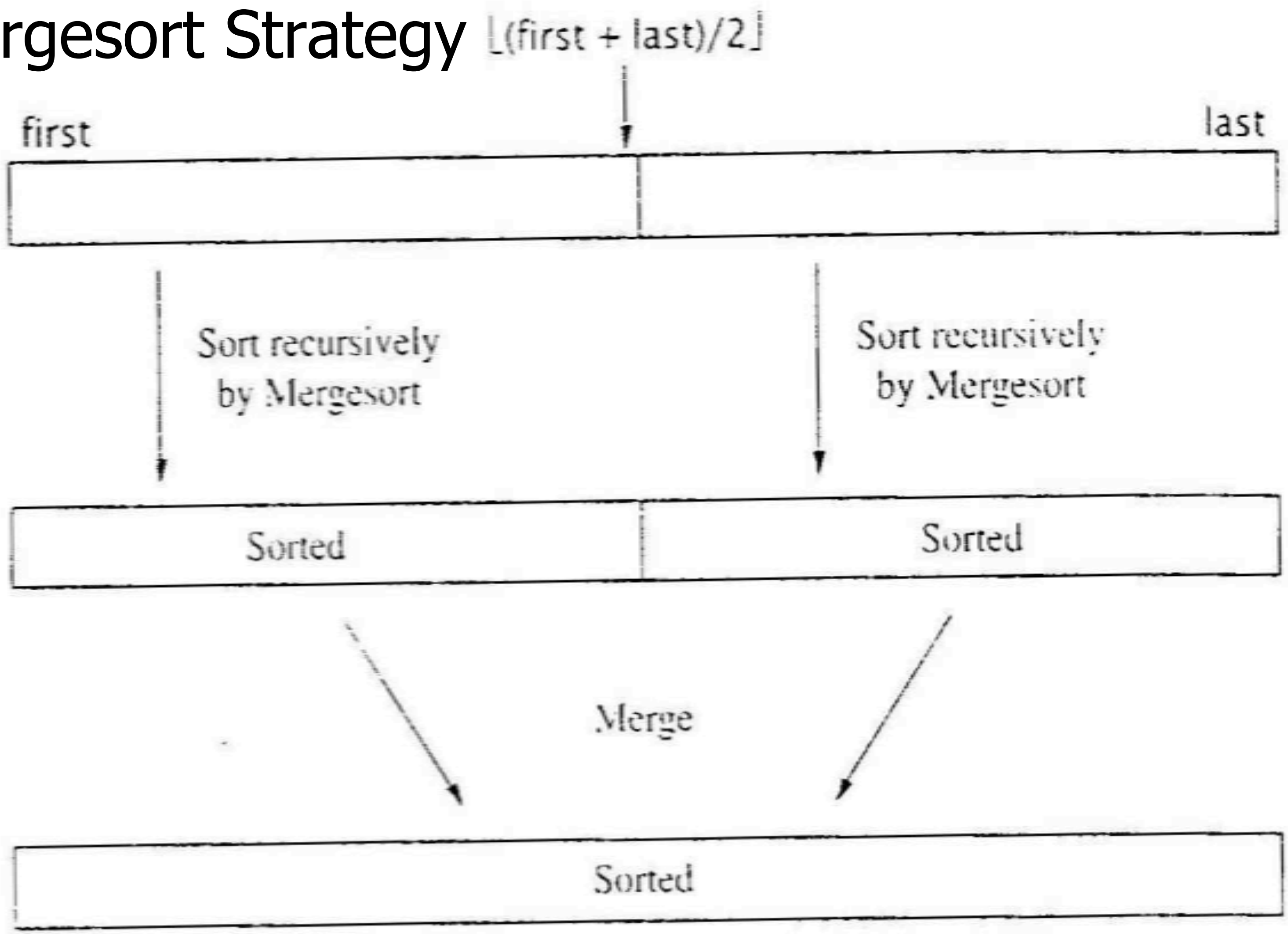
- The problem:
  - Given a sequence $\mathbf{a_0 \ldots a_n}$
    reorder them into a permutation $\mathbf{a'_0 \ldots a'_n}$
    such that $\mathbf{a'_i <= a'_{i+1}}$ for all pairs
    - Specifically, this is sorting in non-descending order…
- Basic operation
  - Comparison of keys.  Why?
    - Controls execution, so total operations often proportional
    - Important for definition of a solution
    - Often an expensive operation (say, large strings are keys)
  - However, swapping items is often expensive
    - We can apply same techniques to count swapping in a separate analysis

# Why Do We Study Sorting?

- An important problem, often needed
  - Often users want items in some order
  - Required to make many other algorithms work well. Example: For searching on sorted data by comparing keys, optimal solutions require $\theta(\log n)$ comparisons using binary search
- And, for the study of algorithms…
  - A history of solutions
  - Illustrates various design strategies and data structures
  - Illustrates analysis methods
  - Can prove something about optimality

# Mergesort is Classic Divide & Conquer

- Mergesort Strategy



# Algorithm: Mergesort

- Specification:
  - Input: Array list and indexes first, and Last, such that the elements list[i] are defined for first <= i <= last.
  - Output: list[first], ..., list[last] is sorted rearrangement of the same elements
- Algorithm:

```
def mergesort(list, first, last):
    if first < last:
        mid = (first+last)/2
        mergesort(list, first, mid)
        mergesort(list, mid+1, last)
        merge(list, first, mid, last)
    return
```

# Exercise: Trace Mergesort Execution

- Can you trace MergeSort() on this list?

```
A = {8, 3, 2, 9, 7, 1, 5, 4};
```

# Efficiency of Mergesort

- Cost to divide in half? No comparisons
- Two subproblems: each size n/2
- Combining results? What is the cost of merging two lists of size n/2
  - Soon we'll see it's n-1 in the worst-case
- Recurrence relation:

$$W(1) = 0$$
$$W(n) = 2\,W(n/2) + Wmerge(n)$$
$$= 2\,W(n/2) + n-1$$

You can now show that this $W(n) \in \theta(n \log n)$

# Merging Sorted Sequences

- Problem:
  - Given two sequences A and B sorted in non-decreasing order, merge them to create one sorted sequence C
  - Input size:  C has n items, and A and B each have n/2
- Strategy:
  - determine the first item in C: It is the minimum between the first items of A and B. Suppose it is the first items of A. Then, rest of C consisting of merging rest of A with B.

# Algorithm: Merge

```
Merge(A, B, C)  // where A, B, and C are sequences
    if (A is empty)
        rest of C = rest of B
    else if (B is empty)
        rest of C = rest of A
    else if (first of A <= first of B)
        first of C = first of A
        merge (rest of A, B, rest of C)
    else
        first of C = first of B
        merge (A, rest of B, rest of C)
    return
```

- **W(n) = n – 1**

# More on Merge, Sorting,…

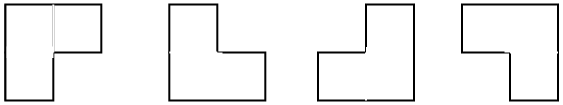
- See Algorithms text, pp. 220-1, for more detailed code for merge
  - See Python example on course-website

- In-place merge is possible (see text)
  - What's "in-place" mean?
  - Space usage is constant, or $\Theta(1)$

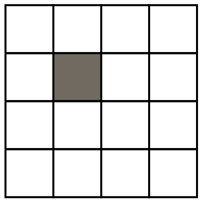
- When is a sort stable?
  - If duplicate keys, their relative order is the same after sorting as it was before
  - Sometimes this is important for an application
  - Why is mergesort stable?

# Next Example: Trominos

- Tiling problems
  - For us, a game: Trominos
  - In “real” life: serious tiling problems regarding component layout on VLSI chips
- Definitions
  - Tromino



  - A deficient board
    - n x n where $n = 2^k$
    - exactly one square missing



- Problem statement:
  - Given a deficient board, tile it with trominos
    - Exact covering, no overlap

# Trominos: Playing the Game, Strategy

- Java app for Trominos: http://www3.amherst.edu/~nstarr/puzzle.html

- How can we approach this problem using Divide and Conquer?
- Small solutions: Can we solve them directly?
  - Yes: 2 x 2 board
- Next larger problem: 4 x 4 board
  - Hmm, need to divide it
  - Four 2 x 2 boards
  - Only one of these four has the missing square
    - Solve it directly!
  - What about the other three?

# Trominos: Key to the Solution

- Place one tromino where three 2 x 2 boards connect
  - You now have three 2 x 2 deficient boards
  - Solve directly!
- General solution for deficient board of size n
  - Divide into four boards
  - Identify the smaller board that has the removed tile
  - Place one tromino that covers the corner of the other three
  - Now recursively process all four deficient boards
  - Don't forget! First, check for n==2

```
Input Parameters: n, a power of 2 (the board size);
                  the location L of the missing square
Output Parameters: None
tile(n,L) {
   if (n == 2) {
      // the board is a right tromino T
      tile with T
      return
   }
   divide the board into four n/2 × n/2 subboards
   place one tromino as in Figure 5.1.4(b)
   // each of the 1 × 1 squares in this tromino
   // is considered as missing
   let m1,m2,m3,m4 be the locations of the missing squares
   tile(n/2,m1)
   tile(n/2,m2)
   tile(n/2,m3)
   tile(n/2,m4)
}
```

# Trominos: Analysis

- What do we count?  What's the basic operation?
  - Note we place a tromino and it stays put
  - No loops or conditionals other than placing a tile
  - Assume placing or drawing a tromino is constant
  - Assume that finding which subproblem has the missing tile is constant
- Conclusion: we can just count how many trominos are placed
- How many fit on a n x n board?
  - $(n^2 - 1) / 3$
- Do you think this optimal?

# Problem: Find Closest Pair of Points

- Given a set of points in 2-space, find a pair that has the minimum distance between them
  - Distance is Euclidean distance
- A computational geometry problem…
  - And other applications where distance is some similarity measure
  - Pattern recognition problems
    - Items identified by a vector of scores
  - Graphics
  - VLSI
  - Etc.

# Obvious Solution: Closest Pair of Points

- For the complete set of n(n-1)/2 pairings, calculate the distances and keep the smallest
  - $\Theta(n^2)$

# An aside: k Nearest Neighbors problem

- How to find the “k nearest neighbors” of a given point X?
  - Pattern recognition problem
  - All points belong to a category, say “cancer risk” and “not at risk”.
  - Each point has a vector of size n containing values for some set of features
  - Given an new unclassified point, find out which category it is most like
  - Find its k nearest neighbors and use their classifications to decide (i.e. they “vote”)
  - If k=1 then this is the closest point problem for n=2

# Solving k-NN problem

- Obvious solution:
  - Calculate distance from X to all other points
  - Store in a list, sort the list, choose the k smallest
- Better solution, better data structure? (Think back to CS2150)
  - Keep a max-heap with the k smallest values seen so far
  - Calculate distance from X to the next point
  - If smaller than the heap's root, remove the root and insert that point into the heap
  - Why a max-heap?

# Back to Closest Pairs

- How’s it work?
- See class notes (done on board), or the textbook

# Summary of the Algorithm

- Strategy:
  - Sort points by x-coordinate
  - Divide into two halves along x-coordinate.
  - Get closest pair in first-half, closest-pair in second-half. Let **d** be value of the closest of these two.
    - In recursion, if 3 points or fewer, solve directly to find closest pair.
  - Gather points in strip of width **2d** into an array **v**
  - For each point in **v**
    - Look at the next 7 points in **v** to see if they closer than **d**

# Analysis: Closest Pairs

- What are we counting exactly?
  - Several parts of this algorithm. No single basic-operation for the whole thing

- (1) Sort all points by x-coordinate: $\Theta(n \lg n)$
- (2) Recurrence: $T(3) = k$
  $T(n) = 2T(n/2) + cn$
  - Checking the strip is clearly $O(n)$
- This is Case 2 of the Main Theorem, so the recursive part is also $\Theta(n \lg n)$

# Matrix Multiplication

- We known how to multiply matrices for a long time!
  - If we count how many arithmetic operations, then it takes $n^3$ multiplications and $n^3$ additions
  - So $\Theta(n^3)$ is “normal”, but could we do better.
  - Hard to see how….
- But matrices and can be broken up into sub-matrices and operated on
  - See pages 233-234 in text book
  - Leads to recursive way to multiply matrices
- One approach: $T(n) = 8T(n/2) + n^2$

# Strassen's Matrix Multiplication

- In 1969, Strassen found a different approach
  - Mathematicians were surprised
- Look at what his approach calculates on p 233.
- Important fact (for us)
  - Just needs 7 multiplications of n/2 size matrices, not 8
  - Also requires $\Theta(n^2)$ arithmetical operations
  - $T(n) = 7T(n/2) + n^2 = n^{\lg 7} = n^{2.807}$
  - Why? Go back and look at our theorems!
- Not just a theoretical result: useful for n>50
- Better result later: $\Theta(n^{2.375})$

# Divide and  Conquer: Bottom-line

- Powerful technique for a wide array of problems
- Don't let a lot of "extra" work fool you:
  - Sometimes recursive pays off
  - But you need to know when
  - Algorithm analysis!