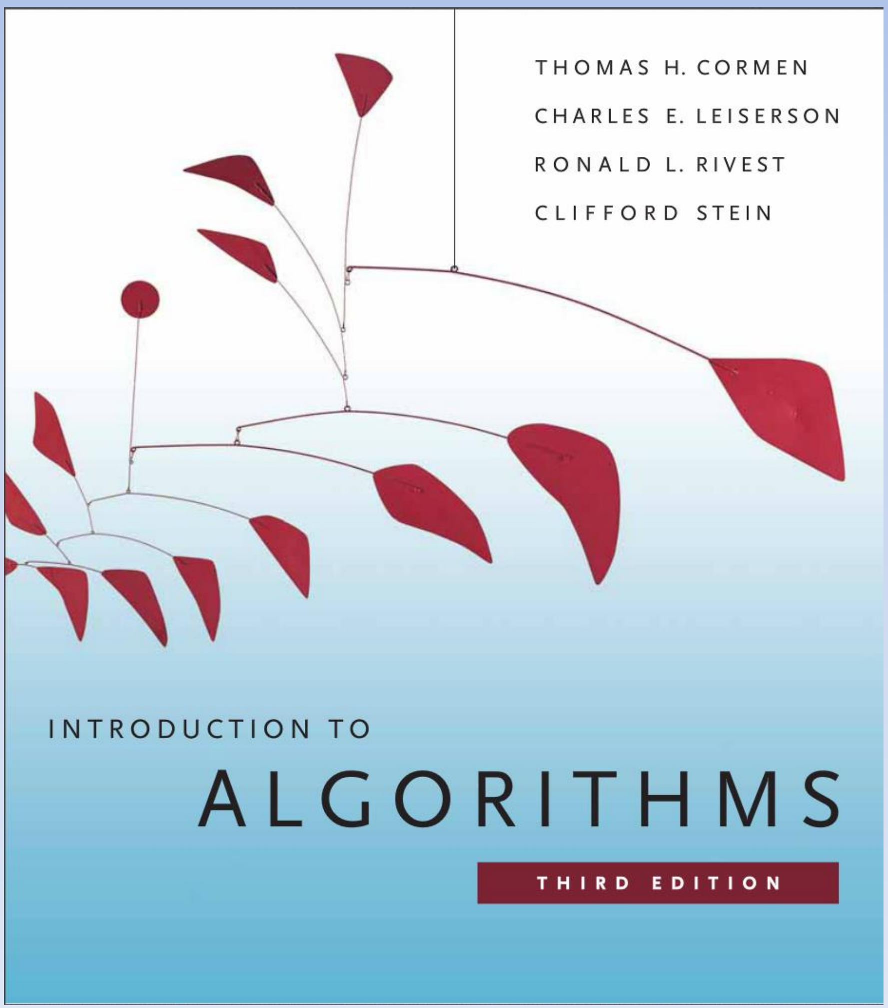
Design and Analysis of Algorithms : Chapter 4

Topic: Divide & Conquer



Divide-and-Conquer

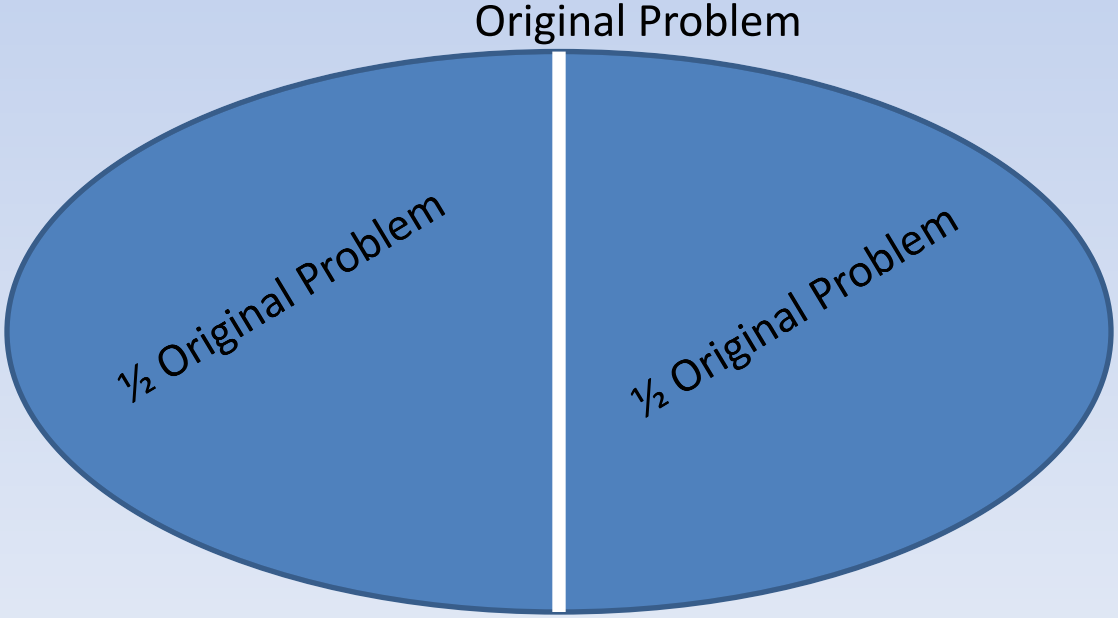
- **Divide** problem into some number of smaller instances of the same problem.
- **Conquer** the subproblems recursively until small enough to just solve.
- **Combine** the subproblem solutions into final solution.

Recurrences w/ Divide & Conquer

- Recurrences define functions recursively
- Recurrence describes function in terms of value on smaller inputs.

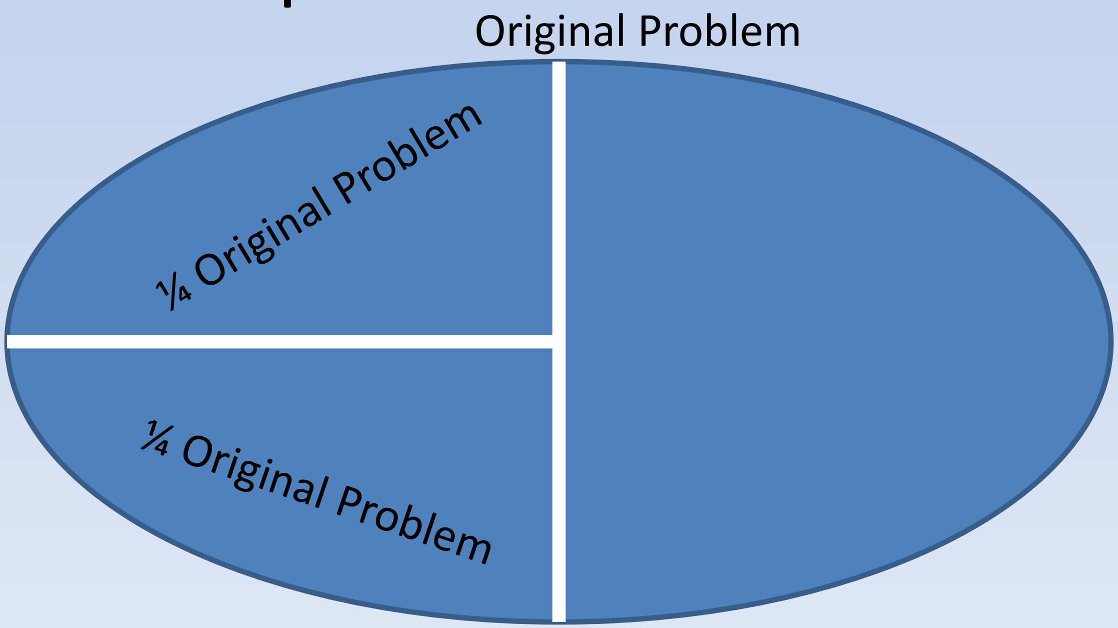
Divide & Conquer

- Break the problem into two equal parts.
- Solve these parts recursively
  - Down to some base case
- Combine two partial solutions



Divide & Conquer

- Break the problem into two equal parts.
- Solve these parts recursively
  - Down to some base case
- Combine two partial solutions



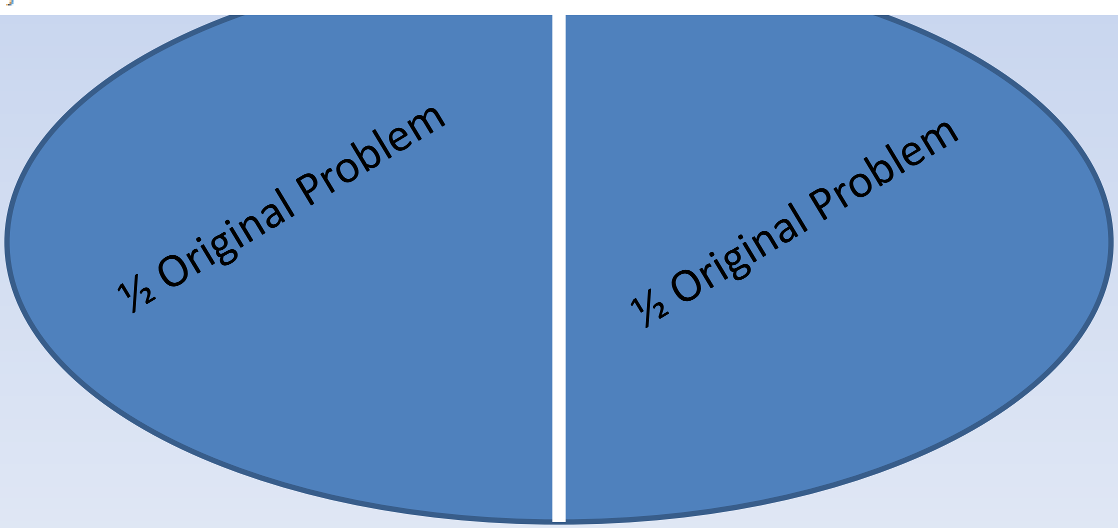
Divide & Conquer: findMax

- Brea
- Solv
  - D
- Com

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

int findMax(int i, int j, vector<int>& A){
  int mid;
  int leftMax, rightMax;
  if (j<=i){
    return A.at(i);
  }
  mid = (i+j)/2;
  leftMax = findMax(i, mid, A);
  rightMax = findMax(mid+1, j, A);
  return max( leftMax, rightMax);
}
```



Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two equal parts
  - Recursive solves each of the parts
  - Combines the two sub-solutions into the final solution doing constant work to combine solutions.
    - Comparing the two max's
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(1), & if\ n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two equal parts
  - Recursive solves each of the parts
  - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two **unequal** parts
    - First part is 2/3 of items
    - Second part is 1/3 of items
  - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

Divide & Conquer Example: The Maximum-Subarray Problem

- Buy Low/Sell High
- How much could I have made???

Divide & Conquer Example: The Maximum-Subarray Problem

MAX RANGE SUM

CHALLENGE DESCRIPTION:

Bob is developing a new strategy to get rich in the stock market. He wishes to invest his portfolio for 1 or more days, then sell it at the right time to maximize his earnings. Bob has painstakingly tracked how much his portfolio would have gained or lost for each of the last N days. Now he has hired you to figure out what would have been the largest total gain his portfolio could have achieved.

For example:

Bob kept track of the last 10 days in the stock market. On each day, the gains/losses are as follows:

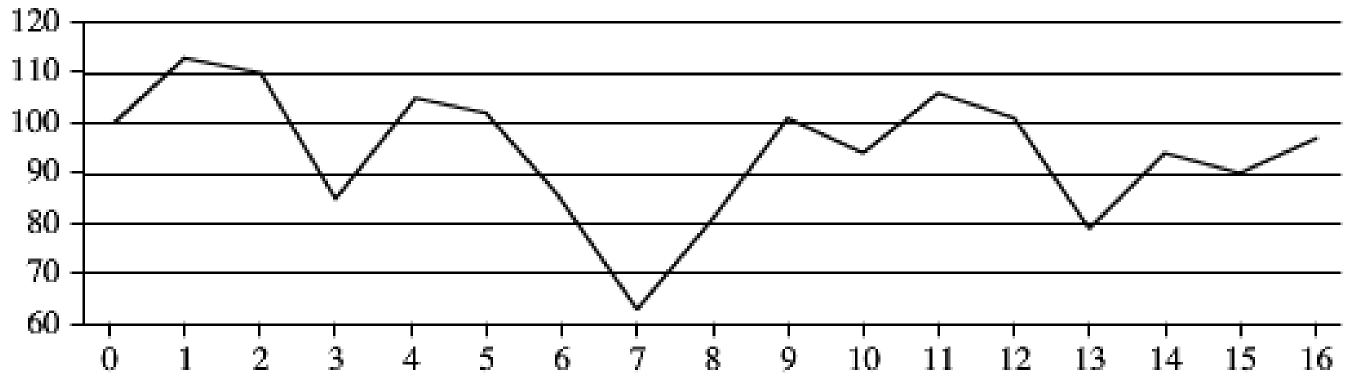
```
7 -3 -10 4 2 8 -2 4 -5 -2
```

If Bob entered the stock market on day 4 and exited on day 8 (5 days in total), his gains would have been

```
16 (4 + 2 + 8 + -2 + 4)
```

- Buy Low/Sell High
- How much could I have made???

Divide & Conquer Example: The Maximum-Subarray Problem



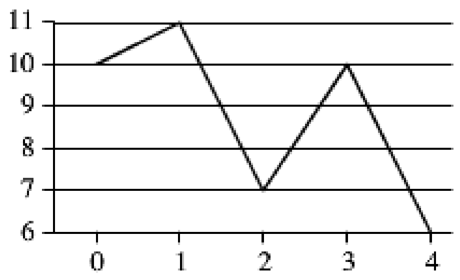
Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

- Buy Low/Sell High
- How much could I have made???

NOT:
Buy Lowest/Sell Highest

4.1 The maximum-subarray problem *69*



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

- Not as simple as buying at lowest & selling at highest!

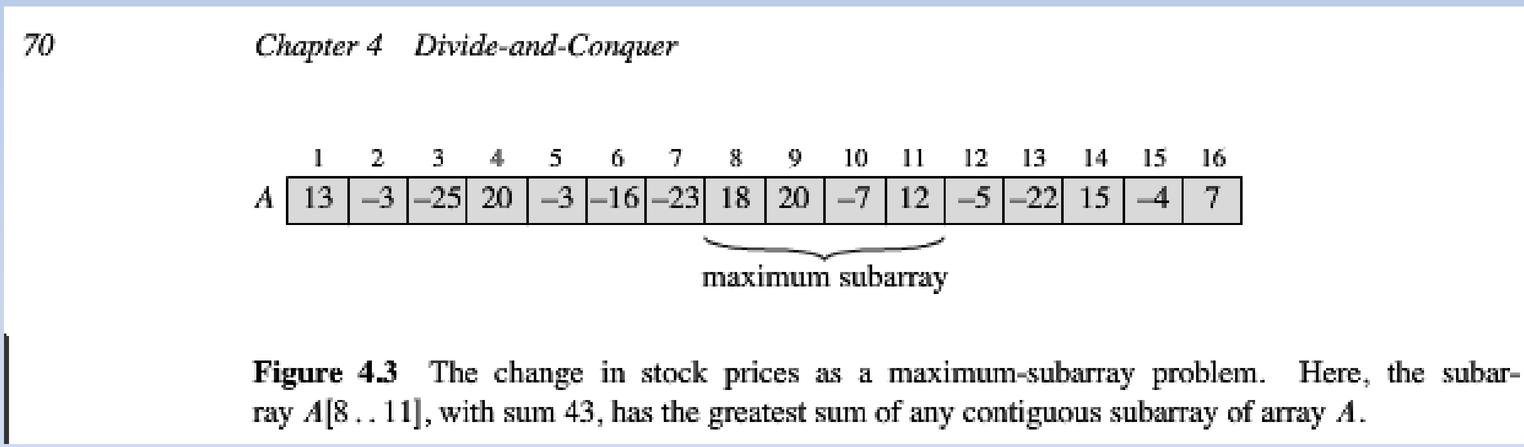
Where do I begin?

- Simple Solution frequently possible
  - Brute-Force
- Try every possible buy and sell date:
  - Number of dates = n
  - Number of buy and sell dates =
    - $\binom{n}{2} = {}^{n(n-1)}/_{2} = \boldsymbol{\Theta}(n^2)$

- Can I Do Better???

Consider Transformation

- Want to develop an algorithm $o(n^2)$
  - Meaning strictly smaller than n^2
- To help: Look @ daily change in price
  - NOT daily price



70 *Chapter 4 Divide-and-Conquer*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

$\underbrace{18 \quad 20 \quad -7 \quad 12}_{\text{maximum subarray}}$

Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

- Find contiguous subarray whose values have largest sum.
- Maximum-Subarray Problem

Maximum Subarray w/ Divide & Conquer

- Divide our array in half.
- Now, Solution to entire problem must be one of three cases:
  - Entirely in first half
  - Entirely in second half
  - Beginning in first half and ending in second half.
- First Half and Second Half solutions found Recursively.

Maximum Subarray Crossing Midpoint

- Not a smaller version of original problem.
- Subarray *must cross* midpoint.
- Any subarray crossing midpoint must be made of two subarrays:
  - Subarray starting in left and ending at midpoint
  - Subarray starting at midpoint and ending in right.

Crossing Midpoint Subarray

4.1 The maximum-subarray problem 71

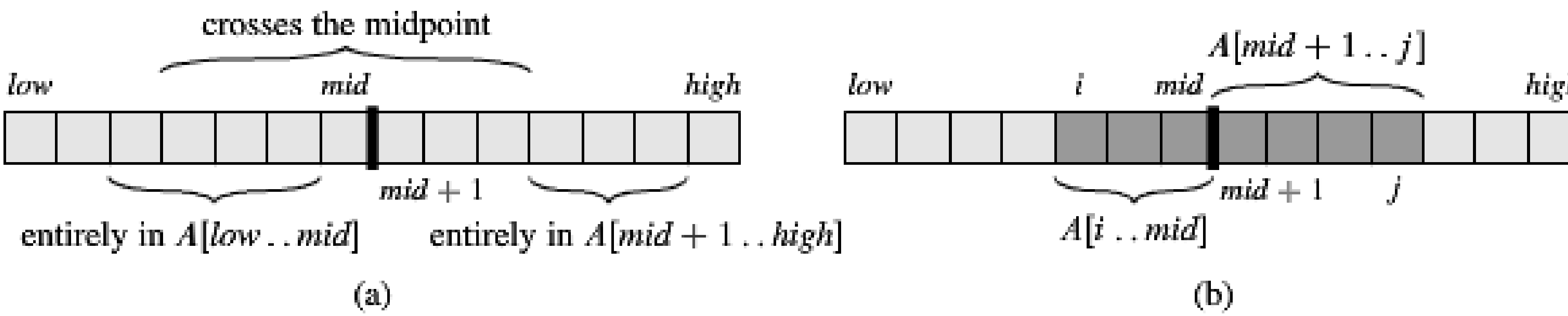


Figure 4.4 (a) Possible locations of subarrays of $A[low \, .. \, high]$: entirely in $A[low \, .. \, mid]$, entirely in $A[mid + 1 \, .. \, high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low \, .. \, high]$ crossing the midpoint comprises two subarrays $A[i \, .. \, mid]$ and $A[mid + 1 \, .. \, j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

- Find maximum subarray ending at mid
- Find maximum subarray starting and mid
- Combine

FIND-MAX-CROSSING-SUBARRAY(*A*, *low*, *mid*, *high*)

```
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Pseudocode

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Analyzing Performance

- Algorithm is Recursive:
  - Establish Recurrence Relation
- Analysis Assumes problem size is power of 2
  - Therefore all subproblem sizes are integers
- Clearly the base case T(1) on line 2 takes constant time
  - Return (low, high, A[low])
  - T(1) = $\boldsymbol{\Theta}(1)$

Pseudocode

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

$\Theta(1)$

Analyzing Performance (2)

- Recursive case given problem size a power of 2:
  - Each of the Two Recursive calls to Find-Maximum-Subarray is applied to a problem size n/2.
  - $T(n/2) + T(n/2) = 2T(n/2)$ required by Lines 4 and 5

Pseudocode

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)                T(n/2)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)           T(n/2)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Pseudocode

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

$\boldsymbol{\Theta}(1)$

$\boldsymbol{\Theta}(1)$

$\boldsymbol{\Theta}(1)$

Analyzing Performance (3)

- $\boldsymbol{\Theta}(n)$ required by Find-Max-Crossing-Subarray

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
```

$\boldsymbol{\Theta}(n/2)$

```
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

$\boldsymbol{\Theta}(n/2)$

Analyzing Performance (4)

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

$\Theta(n)$

Analyzing Performance (5)

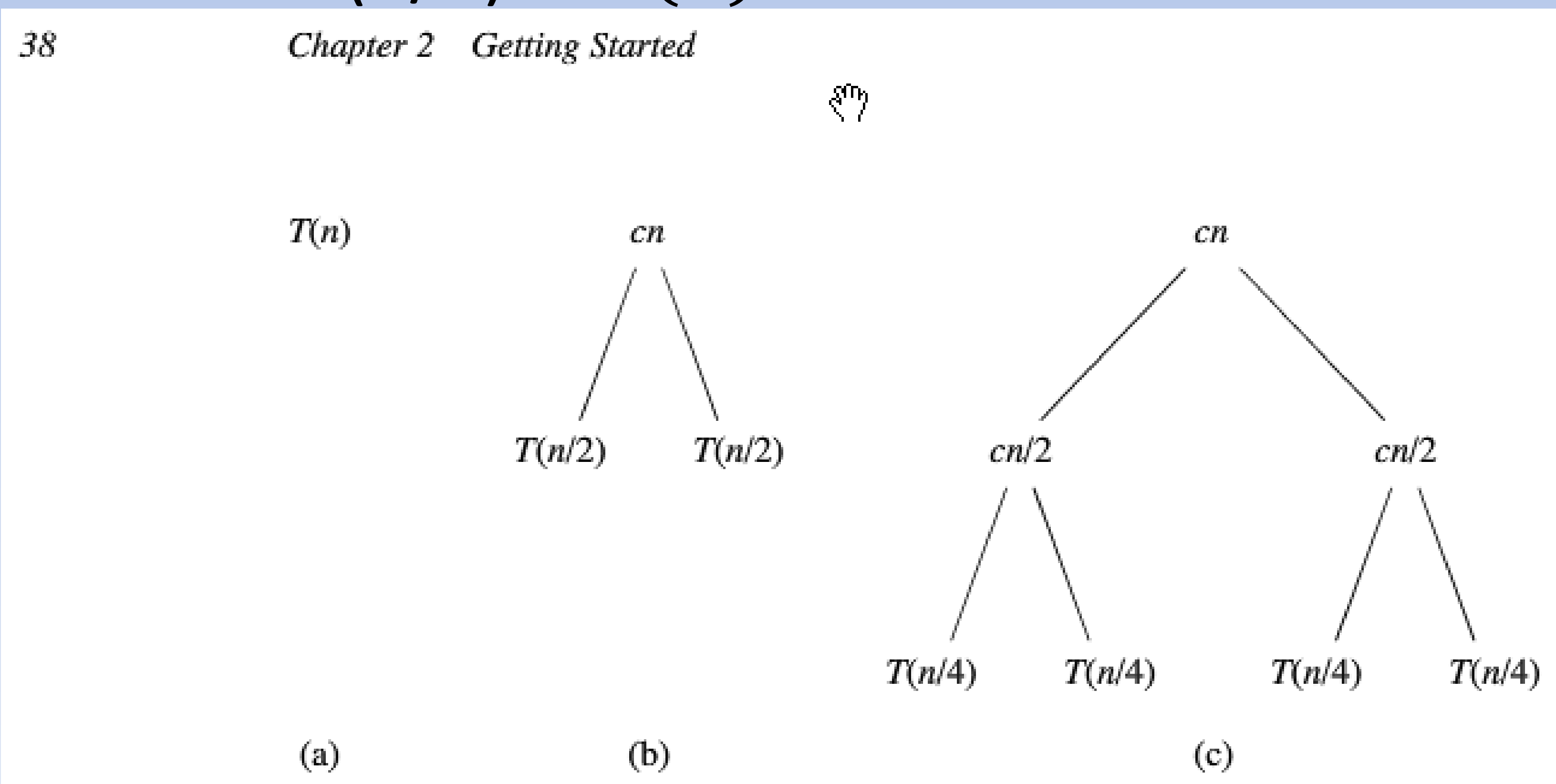
- T(n) = $\boldsymbol{\Theta}(1)$+ 2T(n/2) + $\boldsymbol{\Theta}(n)$ + $\boldsymbol{\Theta}(1)$

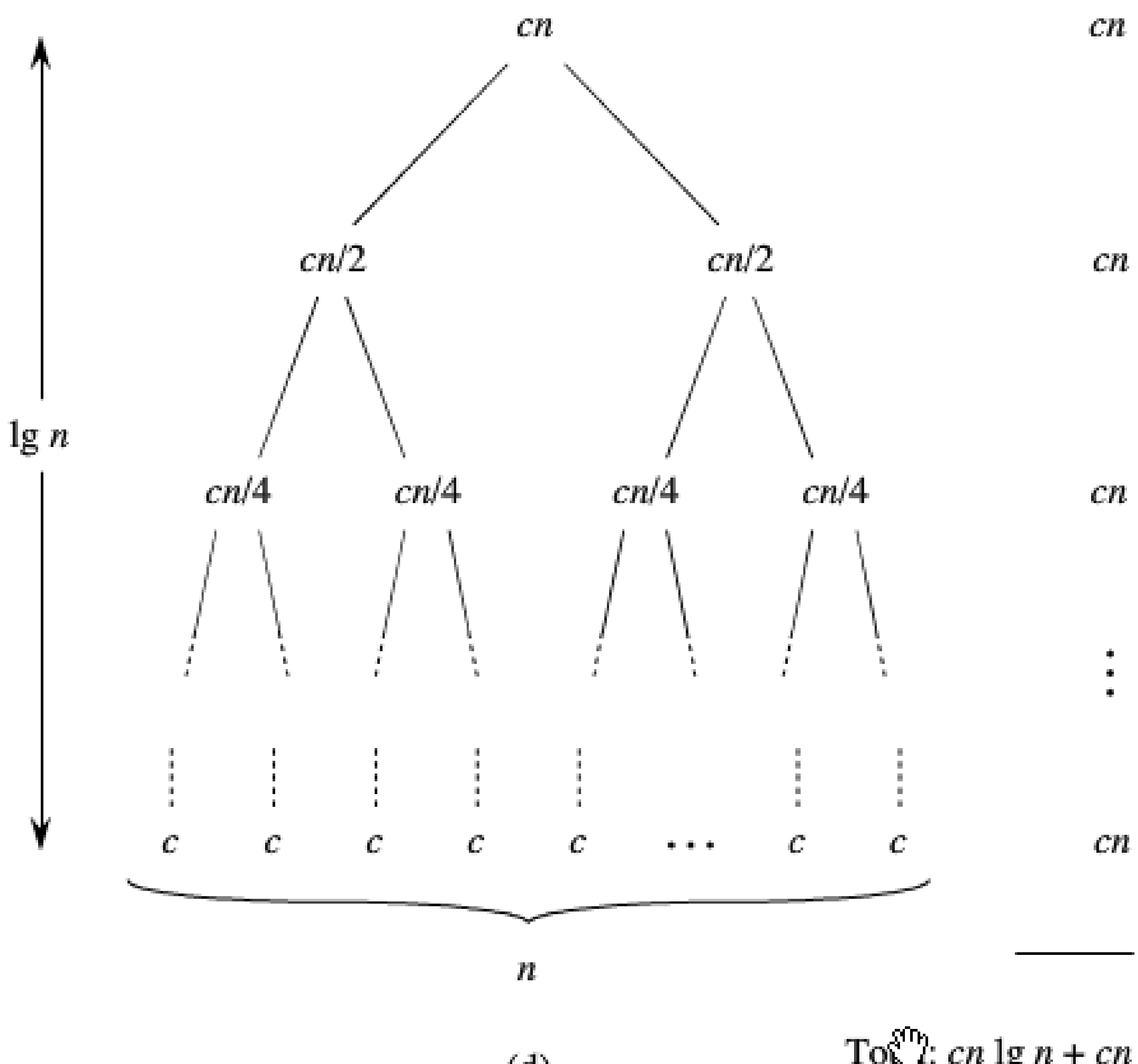
  = 2T(n/2) + $\boldsymbol{\Theta}(n)$

Analyzing Performance (5)

- T(n) = $\boldsymbol{\Theta}(1)$+ 2T(n/2) + $\boldsymbol{\Theta}(n)$ + $\boldsymbol{\Theta}(1)$

  = 2T(n/2) + $\boldsymbol{\Theta}(n)$





cn
cn/2
cn/2
cn/4
cn/4
cn/4
cn/4
c
c
c
c
c
…
c
c
lg n
n
cn
cn
cn
cn
cn
To : cn lg n + cn

(d)

Analyzing Performance (5)

- Model final recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

- **T(n) =** $\boldsymbol{\Theta}(n\ Lg\ n)$

Divide & Conquer Summary

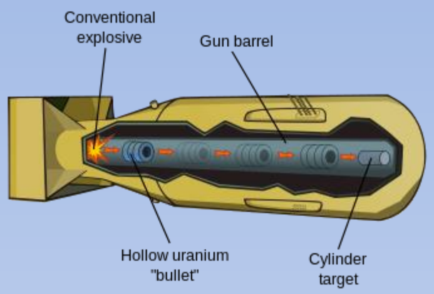
- Split the problem in half and recursively solve smaller versions
  - Creating a Lg(n) recursion tree
- Merge sub solutions in $\boldsymbol{\Theta}(n)$
  - Each level of recursion has the same n items
  - Depth of recursion is $\boldsymbol{Lg\ n}$
- Smart Merge Procedure needed!

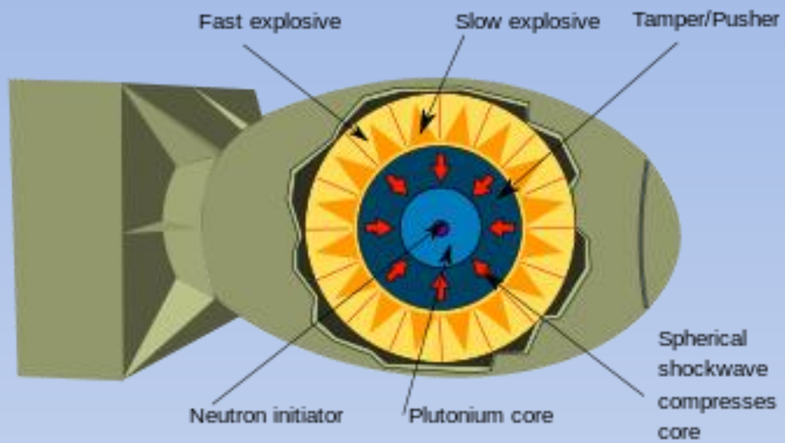
Divide & Conquer w/ Mergesort

- Developed by John von Neumann in 1945

- Allowed large data sets to be sorted on early computers w/ small memories by modern standards.
- Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

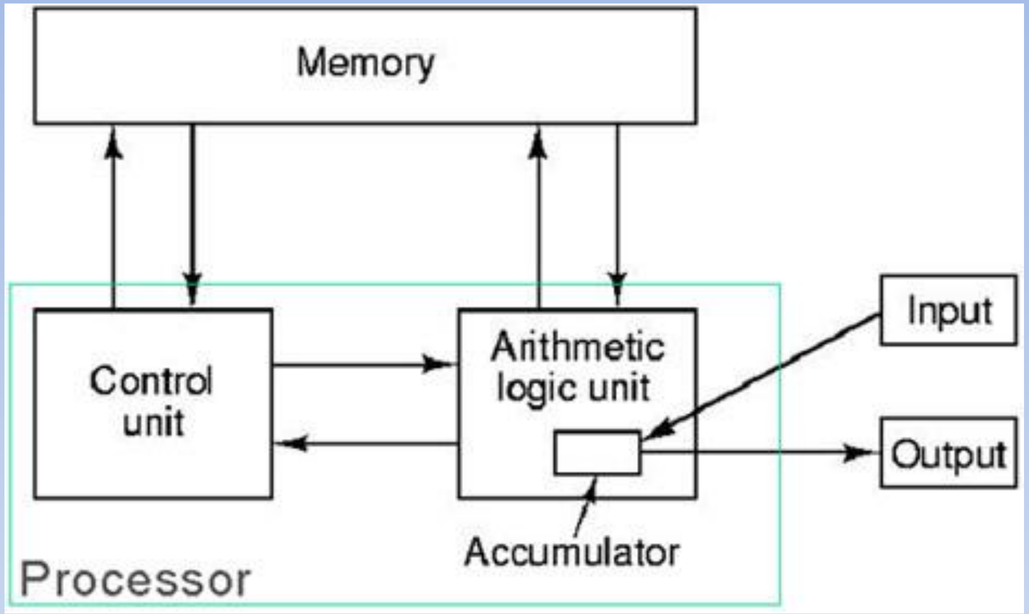
John von Neumann Side Note





- John von Neumann developed an expertise in explosions in the 30's
- Became a leading authority in the area of the mathematics of shaped charges.
- In 1943, von Neumann began working on the Manhattan Project, where he tackled the immense calculations required for construction of an atomic bomb.
- Von Neumann played major role in development of 'implosion' style plutonium atomic bomb used for the first "Trinity" test bomb (July 1945) and "Fat Man" weapon dropped on Nagasaki.
- Immense calculations required for the Atomic work created interest in using machines for the calculation of numbers and the resolution of specific mathematical problems.
- During and after the war, his interest in computers grew, and he contributed extensively to the construction of the first modern computers.

John von Neumann Side Note



- Helped develop modern computer architecture.

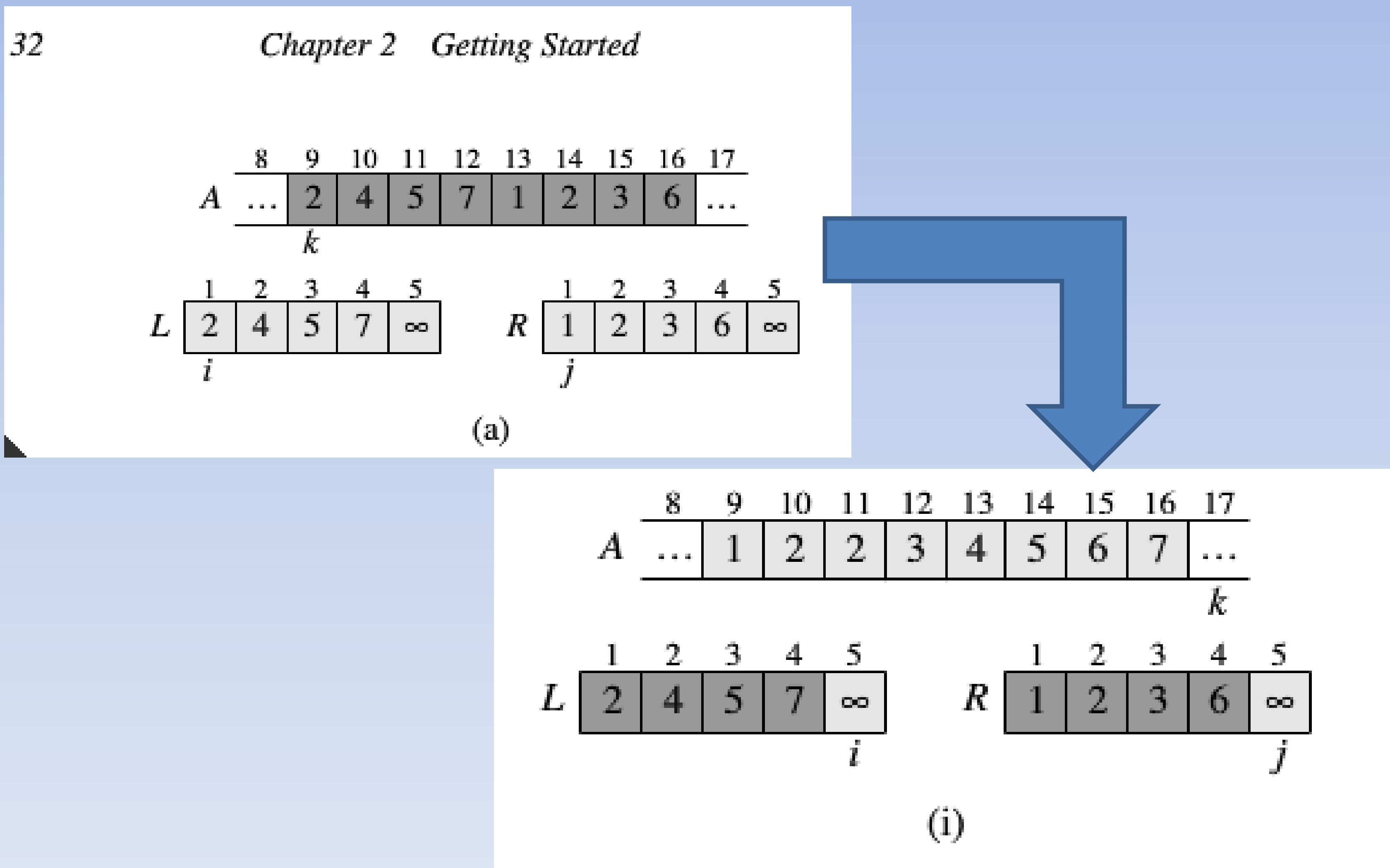
Divide & Conquer w/ Merge sort

- Sort Problem: Take an n-element sequence of numbers and find a permutation where elements are ordered smallest to largest.
- Merge sort:
  - **Divide** the n-element sequence into two subsequence of n/2 elements each.
  - **Conquer** the two subsequences recursively.
  - **Combine** the two sorted subsequences for the final solution.

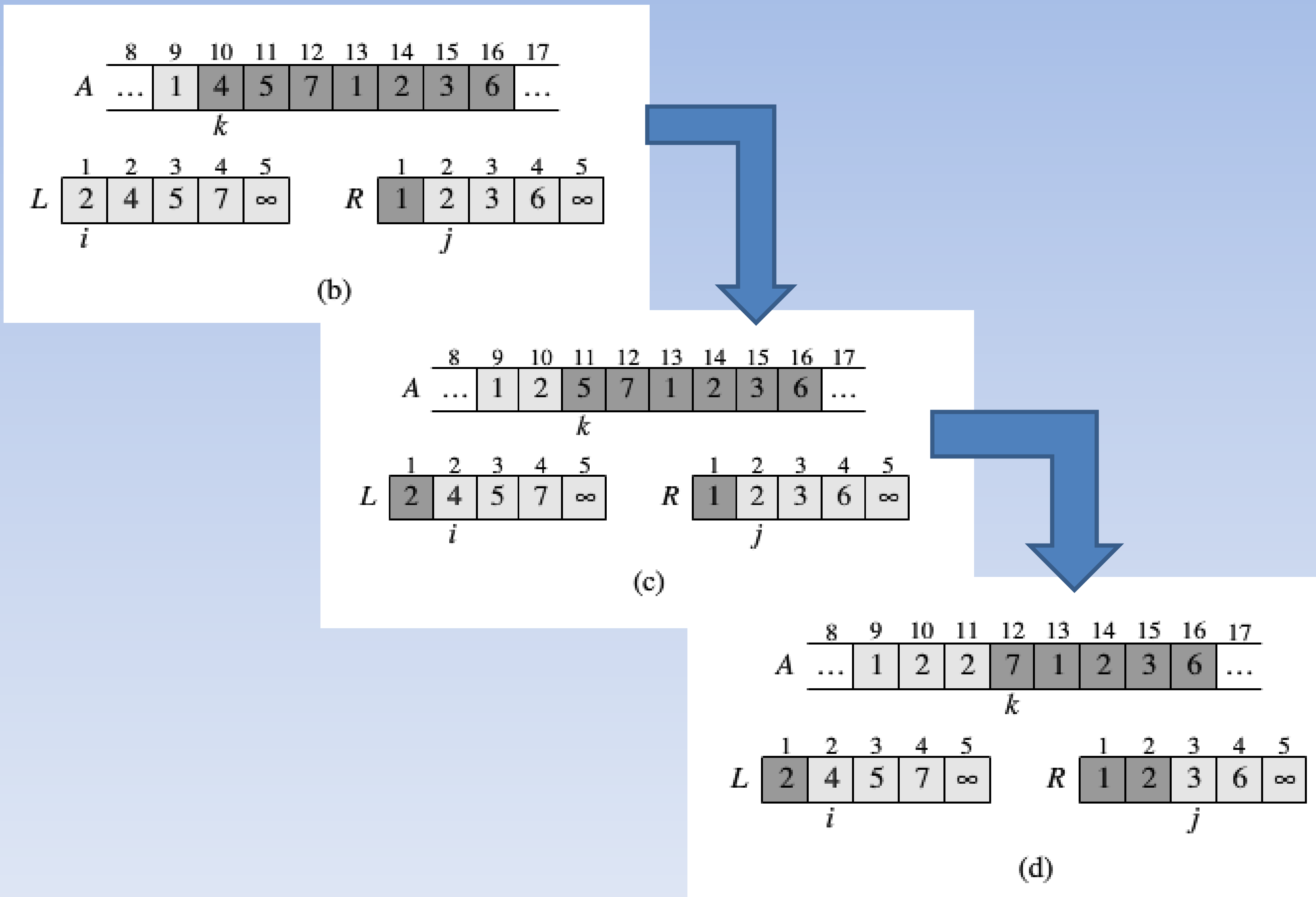
Divide & Conquer w/ Merge sort

- Recursion bottoms out when the sequence is of size 1, which is sorted by definition.
- Key operation is MERGING two sorted sequences.
- Merging is done in time $\boldsymbol{\Theta}(n)$ where n is the total number of items to be merged.

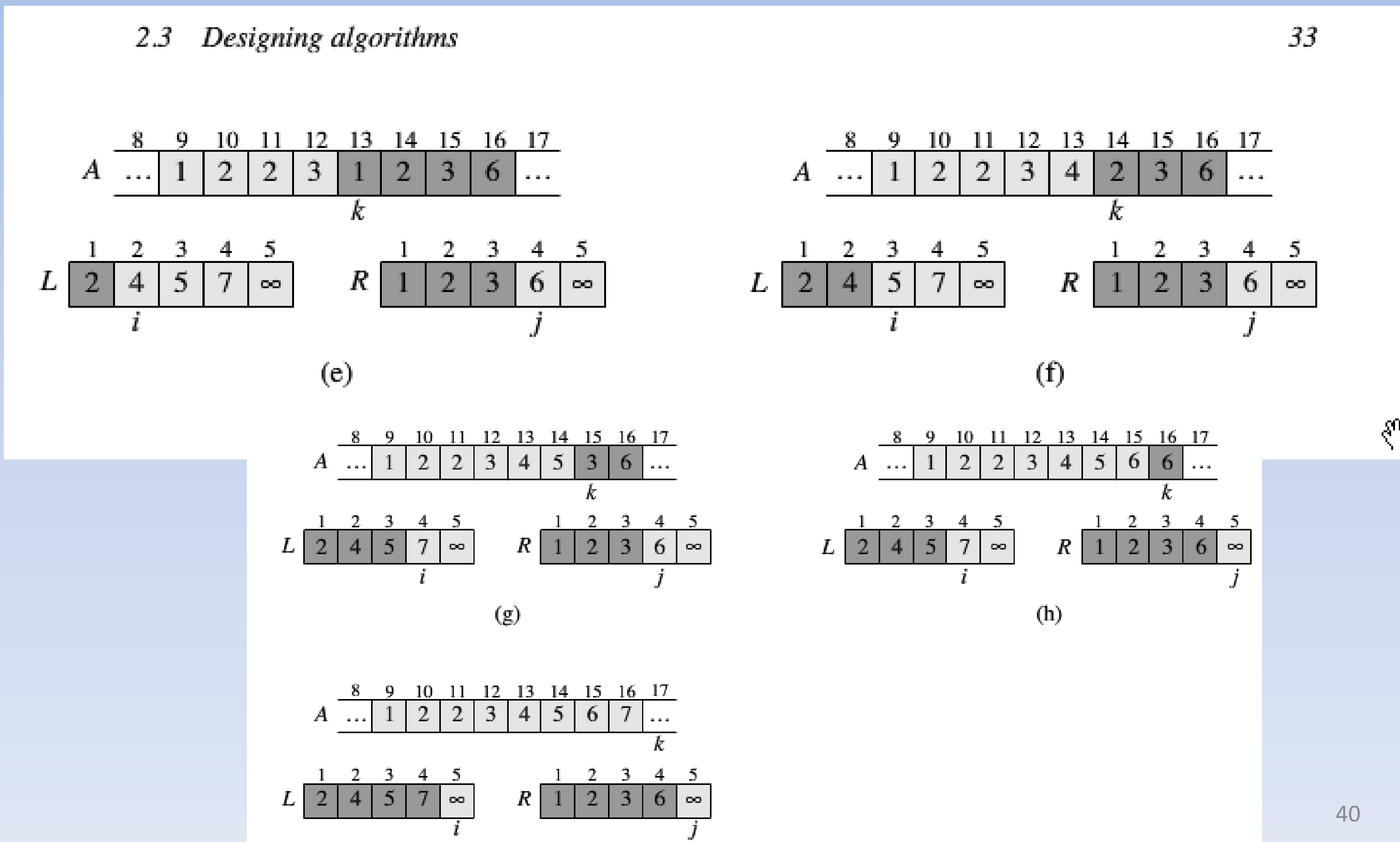
Merging two sorted sequence



Merging two sorted sequence



Merging two sorted sequence



Merge

```
MERGE(A, p, q, r)
n₁ = q − p + 1
n₂ = r − q
let L[1 . . n₁ + 1] and R[1 . . n₂ + 1] be new arrays
for i = 1 to n₁
    L[i] = A[p + i − 1]
for j = 1 to n₂
    R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Copy to Temp

Loop through N Items

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Develop Recurrence Relation
  - Divide step (and base case): $\boldsymbol{\Theta}(1)$
  - Conquer: T(n) = T(n/2) + T(n/2)
  - Combine: Merge procedure is $\boldsymbol{\Theta}(n)$

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Model final recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

- **T(n) =** $\boldsymbol{\Theta}(n\ Lg\ n)$

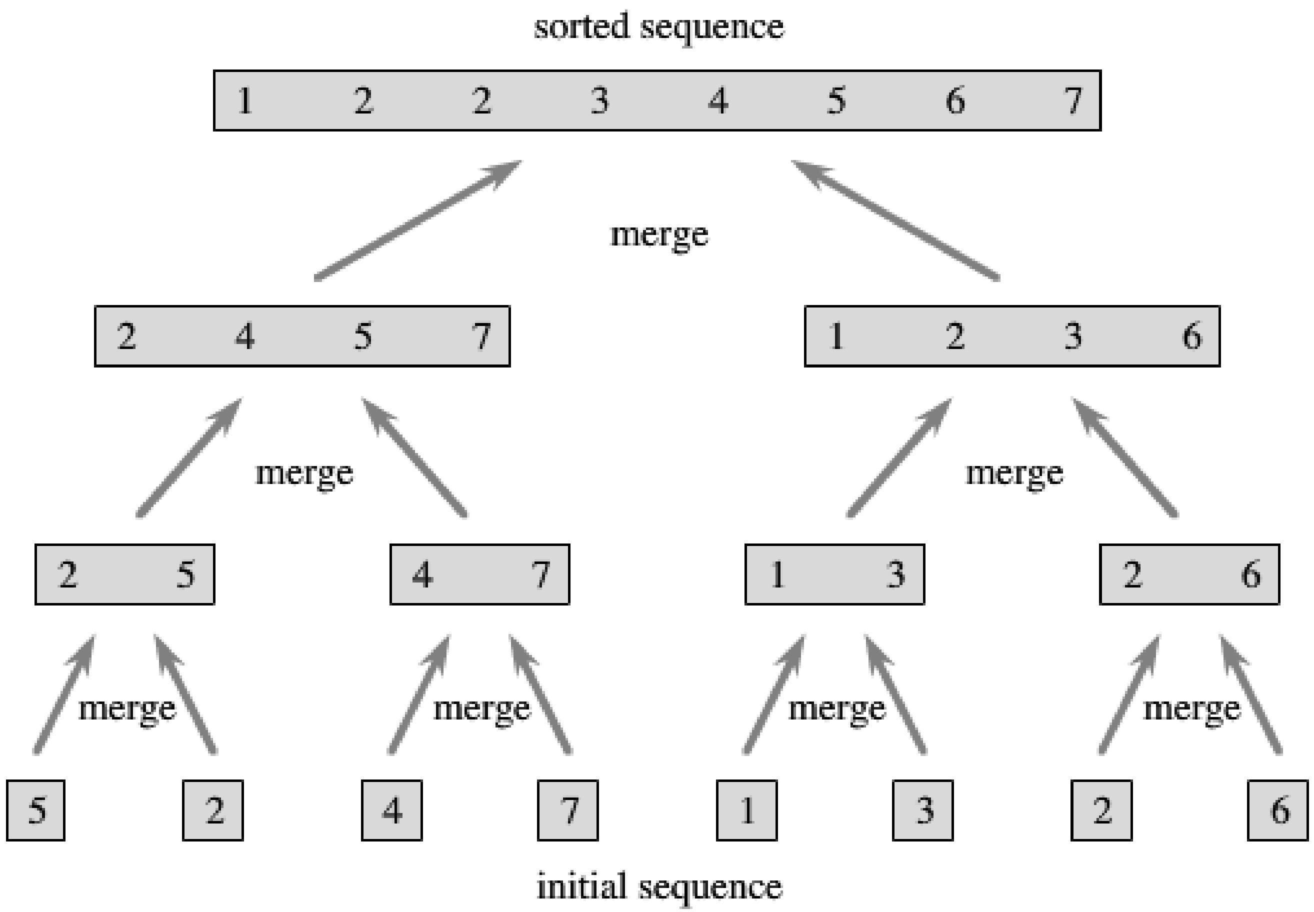
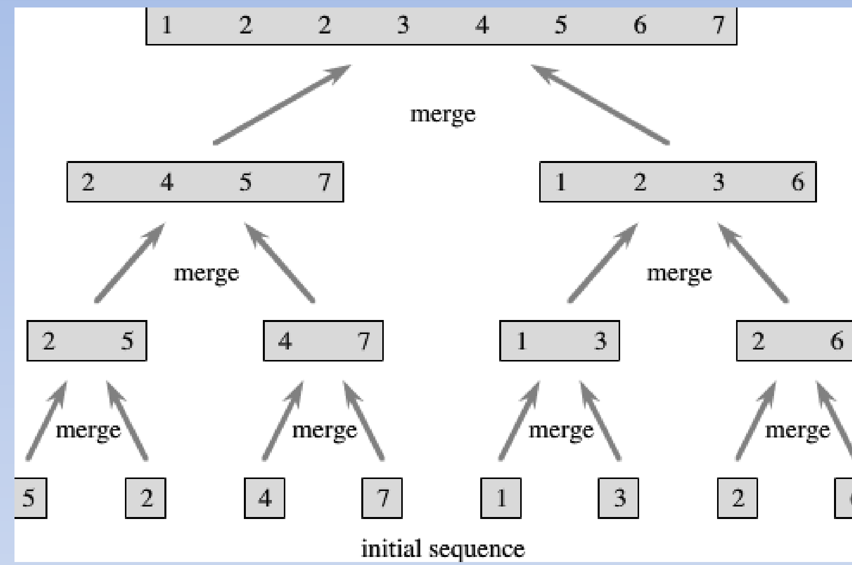


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Look @ Merge Formally

MERGE(A, p, q, r)

```
1   n1 = q - p + 1
2   n2 = r - q
3   let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i - 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```



- Merge done in Lines 12-17 with Invariant:
  - At start of each iteration A[p..k-1] contains k-p smallest elements of L[1..n_1+1] and R[1..n_2+1]
- Must show:
  - Invariant holds before first iteration
  - Invariant maintained by loop
  - Invariant provides useful property when loop terminates

Look @ Merge Formally

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- Merge done in Lines 12-17 with Invariant:
  - At start of each iteration A[p..k-1] contains k-p smallest elements of L[1..n_1+1] and R[1..n_2+1]
- Maintenance:
  - Case L[i] ≤ R[j]: L[i] is smallest element not yet copied back into A.
  - A[p..k-1] currently contains k-p smallest elements, after line 14 copy it will contain k-p+1 smallest elements.
  - Case R[j] ≤ L[j] similarly
  - Incrementing k and i reestablishes the loop invariant for next iteration.

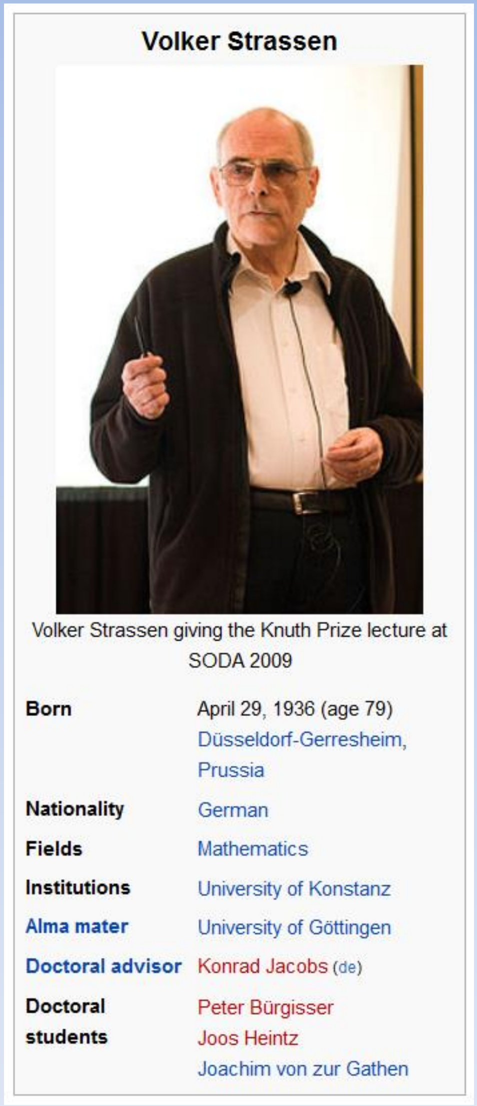
Look @ Merge Formally

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- Merge done in Lines 12-17 with Invariant:
  - At start of each iteration A[p..k-1] contains k-p smallest elements of L[1..n_1+1] and R[1..n_2+1]
- Termination:
  - K=r+1. By loop invariant=> A[p..k-1]=A[p..r] contains the k-p=r-p+1 smallest elements of L[1..n_1+1] and R[1..n_2+1] in sorted order.
  - n_1+n_2+2 = r-p+3 => So all but 2 largest have been copied back (the sentinals).

Strassen Matrix Multiplication

- First published in 1969
- Improves upon the standard matrix multiplication algorithm $O(n^3)$
- $O(n^{\lg 7}) = n^{2.807355}$



SQUARE-MATRIX-MULTIPLY(A, B)

```
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

- Triple loop: n^3

Simple Divide & Conquer
Break Matrix into 4 Quadrants

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

4.2 Strassen's algorithm for matrix multiplication 77

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?
- Base Case:

$$T(1) = \Theta(1) . \quad (4.15)$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?
- Recursive Case:
  - Partition Matrices: $\boldsymbol{\Theta}(1)$ w/ Indices Calculations
  - Add two n/2 Square Matrices: $(n/2)^2$ elements each
    - $n^2/4 \Rightarrow \boldsymbol{\Theta}(n^2)$
  - 8 Recursive calls of size n/2

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) . \end{aligned} \tag{4.16}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases} \tag{4.17}$$

- We'll look closer at solving this recurrence, but for now:
  - T(n) = $\boldsymbol{\Theta}(n^3)$
- We have NOT done better!!
- Where's the problem??

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

TOO MANY SUBPROBLEMS!!

- We'll look closer at solving this recurrence, but for now:
  - T(n) = $\boldsymbol{\Theta}(n^3)$
- We have NOT done better!!
- Where's the problem??
  - Strassen's Improvement… FEWER SUBPROBLEMS!!!

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

SEVEN Matrix Products!!

- Key Contribution: 7 Recursive Calls!!!
  - NOT 8!!

Recurrence Relation w/ Strassen's Method

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \qquad (4.18)$$

SEVEN Matrix Products!!

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices cre recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. E $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

SEVEN Matrix Products!!

Recurrence Relation w/ Strassen's Method

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \qquad (4.18)$$

SEVEN Matrix Products!!

- Instead of T(n) = $\boldsymbol{\Theta}(n^3)$
  - $\boldsymbol{\Theta}(n^{\log_2 8})$
- For Strassen's Algorithm $\boldsymbol{\Theta}(n^{\log_2 7})$

Strassen's Method w/ 10 Matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

$$\begin{aligned}
S_1 &= B_{12} - B_{22}, \\
S_2 &= A_{11} + A_{12}, \\
S_3 &= A_{21} + A_{22}, \\
S_4 &= B_{21} - B_{11}, \\
S_5 &= A_{11} + A_{22}, \\
S_6 &= B_{11} + B_{22}, \\
S_7 &= A_{12} - A_{22}, \\
S_8 &= B_{21} + B_{22}, \\
S_9 &= A_{11} - A_{21}, \\
S_{10} &= B_{11} + B_{12}.
\end{aligned}$$

Strassen's Method w/ 10 Matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

7 Matrix Multiplies w/ 10 Matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \,, \\
S_2 &= A_{11} + A_{12} \,, \\
S_3 &= A_{21} + A_{22} \,, \\
S_4 &= B_{21} - B_{11} \,, \\
S_5 &= A_{11} + A_{22} \,, \\
S_6 &= B_{11} + B_{22} \,, \\
S_7 &= A_{12} - A_{22} \,, \\
S_8 &= B_{21} + B_{22} \,, \\
S_9 &= A_{11} - A_{21} \,, \\
S_{10} &= B_{11} + B_{12} \,.
\end{aligned}
$$

Strassen's Method w/ 10 Matrices (TRICKY)

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} ,\\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} ,\\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} ,\\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} ,\\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} ,\\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} ,\\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

7 Matrix Multiplies w/ 10 Matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} ,\\
S_2 &= A_{11} + A_{12} ,\\
S_3 &= A_{21} + A_{22} ,\\
S_4 &= B_{21} - B_{11} ,\\
S_5 &= A_{11} + A_{22} ,\\
S_6 &= B_{11} + B_{22} ,\\
S_7 &= A_{12} - A_{22} ,\\
S_8 &= B_{21} + B_{22} ,\\
S_9 &= A_{11} - A_{21} ,\\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

Strassen's Method w/ 7 Sub-Matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

7 Matrix Multiplies w/ 10 Matrices

$$C_{11} = P_5 + P_4 - P_2 + P_6 \,.$$

$$
\begin{array}{l}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
\qquad\qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad\qquad + A_{22} \cdot B_{21} \\
\qquad\quad - A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad\qquad\qquad\qquad - A_{12} \cdot B_{22} \\
\qquad\qquad\qquad\qquad\quad - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + A_{12} \cdot B_{21} \,,
\end{array}
$$

$$C_{12} = P_1 + P_2 \,,$$

and so C_{12} equals

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad\qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad\quad + A_{12} \cdot B_{22} \,, \end{array}$$

corresponding to equation (4.12).

Setting

$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad\qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad\quad + A_{22} \cdot B_{21} \,, \end{array}$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 \,,$$

so that C_{22} equals

$$\begin{array}{l} \quad A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\ \qquad\qquad - A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad\qquad + A_{11} \cdot B_{12} \\ \qquad\qquad\qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad\qquad\qquad\qquad - A_{21} \cdot B_{11} \\ - A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad\qquad\qquad - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12} \\ \hline \qquad\qquad\qquad\qquad\qquad\qquad A_{22} \cdot B_{22} \qquad\qquad\qquad\qquad + A_{21} \cdot B_{12} \,, \end{array}$$

Finding Closest Pair of Points

Finding the closest pair of points w/ Divide & Conquer

- Consider the Problem of finding the two closest pair of points in a set Q of points.
- Brute force will look at all pairs resulting in $\boldsymbol{\Theta}(n^2)$
- Textbook provides an algorithm whose running time is described by our familiar recurrence… T(n) = 2T(n) + O(n)
- Resulting runtime is therefore O(n lg n)

Divide-and-Conquer Algorithm

- Algorithm takes as input:
  - P a subset of points
  - X and Y which contain points sorted by x-coordinate and y-coordinate respectively.

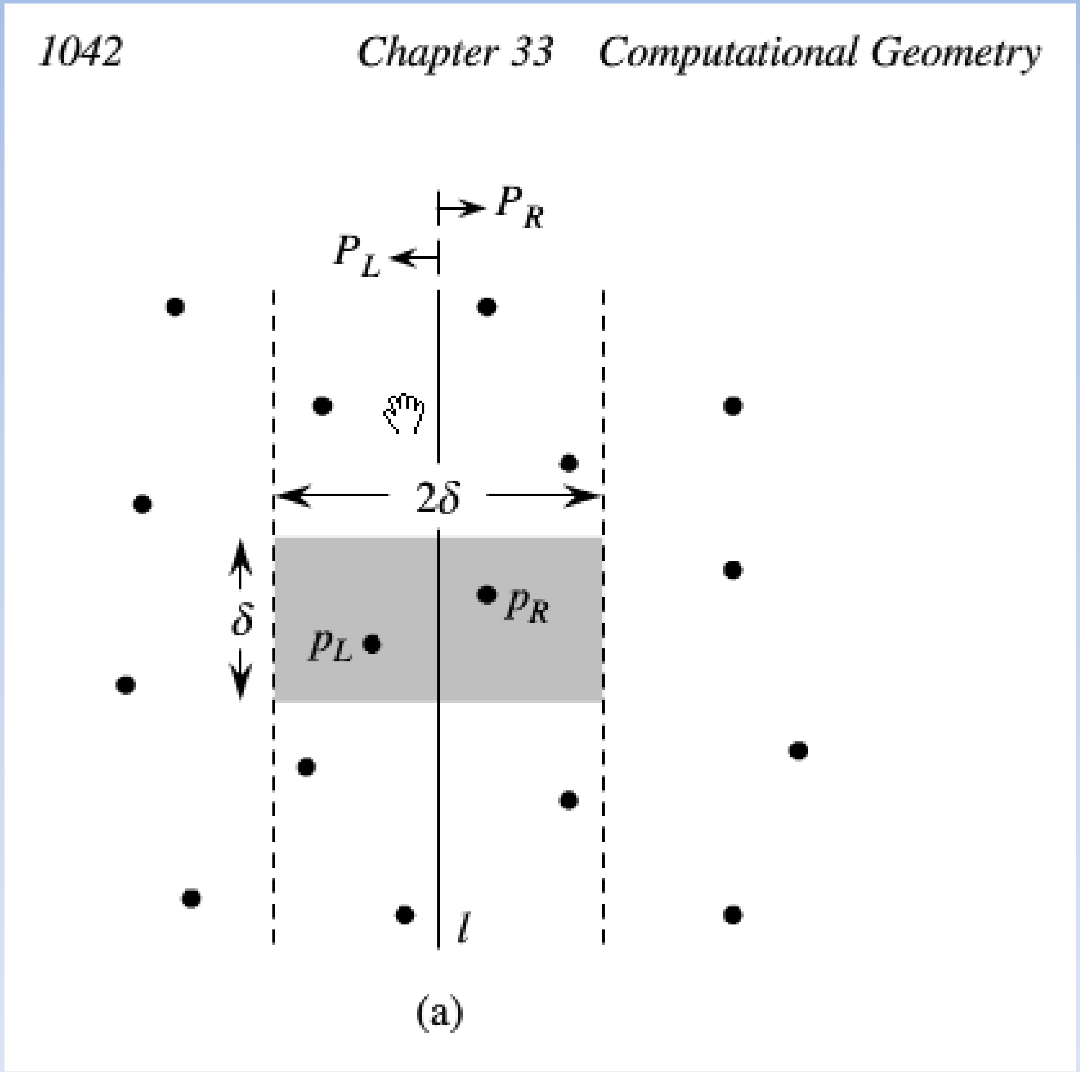
Divide-and-Conquer Algorithm

- Divide:
  - Find a vertical line that bisects the point set P in half, P_l and P_r
  - Divide the X and Y into X_L, X_R, Y_L, Y_R based on whether they are in P_L or P_R maintaining the sorted orders.
- Conquer:
  - Now make two recursive calls with (P_L, X_L, Y_L) and (P_R, X_R, Y_R). Let δ equal the minimum distance of the closest pairs from the two recursive calls.
- Combine:
  - Uses the δ from recursive calls to limit the points tested for possible closest pairs overlapping center.

Combine Step

- Closest pair is either the pair with distance δ found by one of the recursive calls
- OR Closest pair is a pair of points with one point in P_L and the other in P_R.
- Algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ.
- NOTE: if a pair of points has distance less than δ, both points of the pair must be within δ units of line l.
  - Thus residing in the 2δ–wide vertical strip centered at line l.

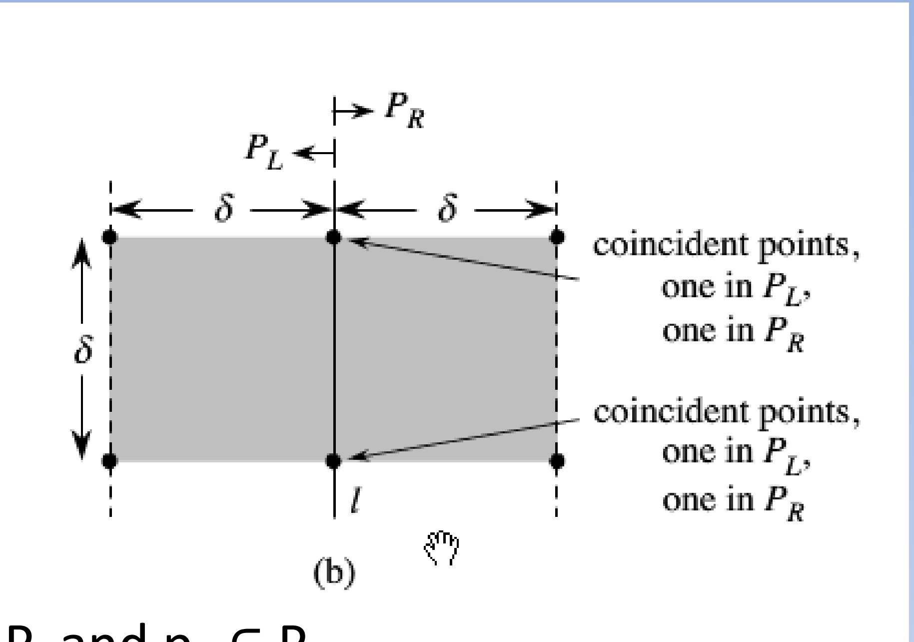
Combine Step



Combine

1. Create array Y’ by removing from Y all but points in 2δ–wide vertical strip leaving array sorted by y-coordinate (like Y)
2. For each point p in array Y’:
   - Try to find points in Y’ that are within δ units of p.
   - Only 7 points that follow p need be considered!
3. If pair of points closer than δ found in step 2 return them, else return closest pair from recursive step.

Checking 7



- Assume closest pair of points i $p_L \in P_L$ and $p_R \in P_R$.
- Distance δ' between p_L and p_R is strictly less than δ.
- Point p_L must be on or to the left of line l and less than δ units away.
- Point p_R is on or to the right of l and less than δ units away.
- Moreover, p_L and p_R are within δ units of each other vertically.

- Points p_L and p_R must be within a δ X 2δ rectangle centered at line l.
- At most 8 points of P can resides within the δ X 2δ rectangle since these points are at least δ units apart.
- Even if p_L occurs as early as possible in Y' and p_R as late as possible, p_R is in one of the 7 positions following p_L.

Running Time

- Goal Recurrence was $T(n) = 2T(n/2) + O(n)$
- Main challenge is sorted arrays X_L, X_R, Y_L, and Y_R
- Method used can be viewed as the opposite of the Merge procedure of merge sort.

Sorted Arrays for Recursive Calls

33.4 Finding the closest pair of points *1043*

Section 2.3.1: we are splitting a sorted array into two sorted arrays. The following pseudocode gives the idea.

```
let Y_L[1 .. Y.length] and Y_R[1 .. Y.length] be new arrays
Y_L.length = Y_R.length = 0
for i = 1 to Y.length
    if Y[i] ∈ P_L
        Y_L.length = Y_L.length + 1
        Y_L[Y_L.length] = Y[i]
    else Y_R.length = Y_R.length + 1
        Y_R[Y_R.length] = Y[i]
```

- Examine points in Y in order
- If a point is in P_L append to Y_L, else append to Y_R
  - Similarly with other sorted arrays.
- Finally sort all points at start of algorithm for an addition n Lg n work.
  - $T(n)' = T(n) + O(n \lg n) = O(n \lg n) + O(n \lg n) = O(n \lg n)$