UMC 201: Data Structures & Algorithms

Lecture 2: Divide-and-conquer: Merge sort, Matrix multiplication; Recurrences

Indian Institute of Science

Recap

- We presented the first example of a sorting algorithm, namely Insertion sort.
- We discussed the RAM model and worst-case running time of an algorithm.

Recap

- We presented the first example of a sorting algorithm, namely Insertion sort.
- We discussed the RAM model and worst-case running time of an algorithm.
- We also introduced the order notations: O, Ω, Θ, o, and ω.

Recap

- We presented the first example of a sorting algorithm, namely Insertion sort.
- We discussed the RAM model and worst-case running time of an algorithm.
- We also introduced the order notations: O, Ω, Θ, o, and ω.

- The worst-case running time of Insertion sort is $\Theta(n^2)$, where n is the number of values to sort.

Divide-and-Conquer

Divide-and-conquer technique

- The divide-and-conquer method is a powerful strategy for designing efficient algorithms.

- Algorithms following the divide-&-conquer method:
  - Divide the problem into *subproblems* that are similar to the original problem but *smaller is size*.

Divide-and-conquer technique

- The divide-and-conquer method is a powerful strategy for designing efficient algorithms.

- Algorithms following the divide-&-conquer method:
  - Divide the problem into *subproblems* that are similar to the original problem but *smaller is size*.
  - Conquer (solve) the subproblems *recursively*.

Divide-and-conquer technique

- The divide-and-conquer method is a powerful strategy for designing efficient algorithms.

- Algorithms following the divide-&-conquer method:
  - Divide the problem into *subproblems* that are similar to the original problem but *smaller is size*.
  - Conquer (solve) the subproblems *recursively*.
  - Combine the solutions *efficiently* to create a solution to the original problem.

Divide-and-conquer technique

- We'll use the divide-and-conquer method to design a sorting algorithm whose worst-case running time is *much less than that of insertion sort*.

Divide-and-conquer technique

- We'll use the divide-and-conquer method to design a sorting algorithm whose worst-case running time is *much less than that of insertion sort*.

- This sorting algorithm is called merge sort.

- We'll show that the worst-case running time of merge sort is $\Theta(n \log n)$ (as opposed to the $\Theta(n^2)$ worst-case running time of insertion sort.)

Merge Sort

Merge sort: The idea

- Start with the array $A[1:n]$ and recurse down to *sort & combine smaller and smaller subarrays* as follows:

Merge sort: The idea

- Start with the array $A[1:n]$ and recurse down to *sort & combine smaller and smaller subarrays* as follows:
  - Divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$, where q is the *midpoint* of $A[p:r]$.

- Initially, $p = 1$ and $r = n$.

Merge sort: The idea

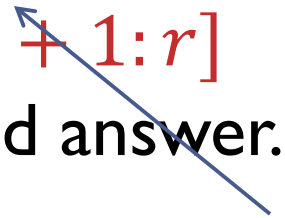
- Start with the array $A[1:n]$ and recurse down to *sort & combine smaller and smaller subarrays* as follows:
  - Divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$, where q is the *midpoint* of $A[p:r]$.
  - Conquer by sorting $A[p:q]$ and $A[q+1:r]$, *recursively* using merge sort.

Merge sort: The idea

- Start with the array $A[1:n]$ and recurse down to *sort & combine smaller and smaller subarrays* as follows:
  - Divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$, where q is the *midpoint* of $A[p:r]$.
  - Conquer by sorting $A[p:q]$ and $A[q+1:r]$, *recursively* using merge sort.
  - Combine by *merging* the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:q]$, producing the sorted answer.

Merge sort: The idea

- Start with the array $A[1:n]$ and recurse down to *sort & combine smaller and smaller subarrays* as follows:
  - Divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$, where q is the *midpoint* of $A[p:r]$.
  - Conquer by sorting $A[p:q]$ and $A[q+1:r]$, *recursively* using merge sort.
  - Combine by *merging* the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:r]$, producing the sorted answer.



Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

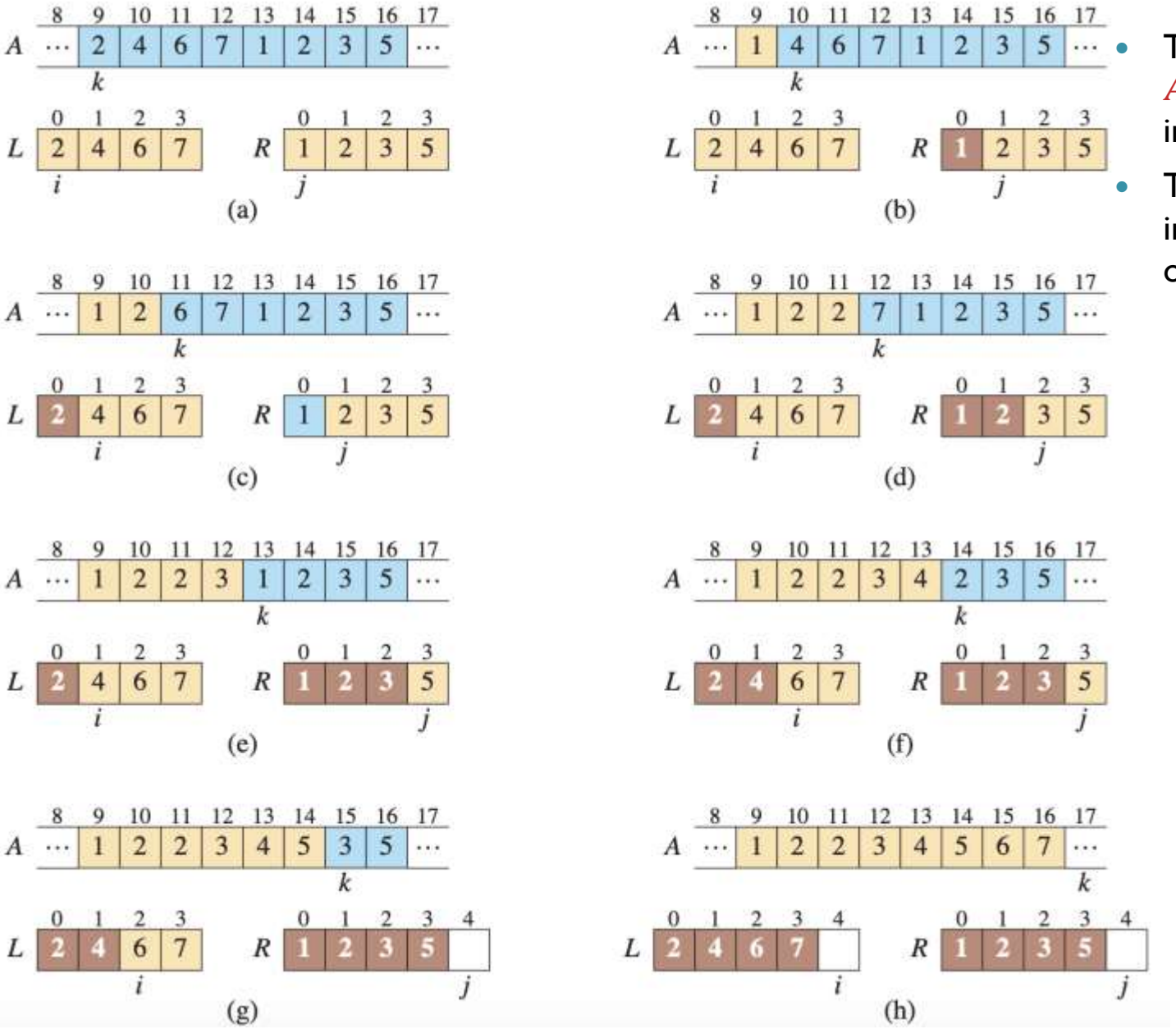
- Invoke the algorithm with $p = 1$ and $r = n$.

Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

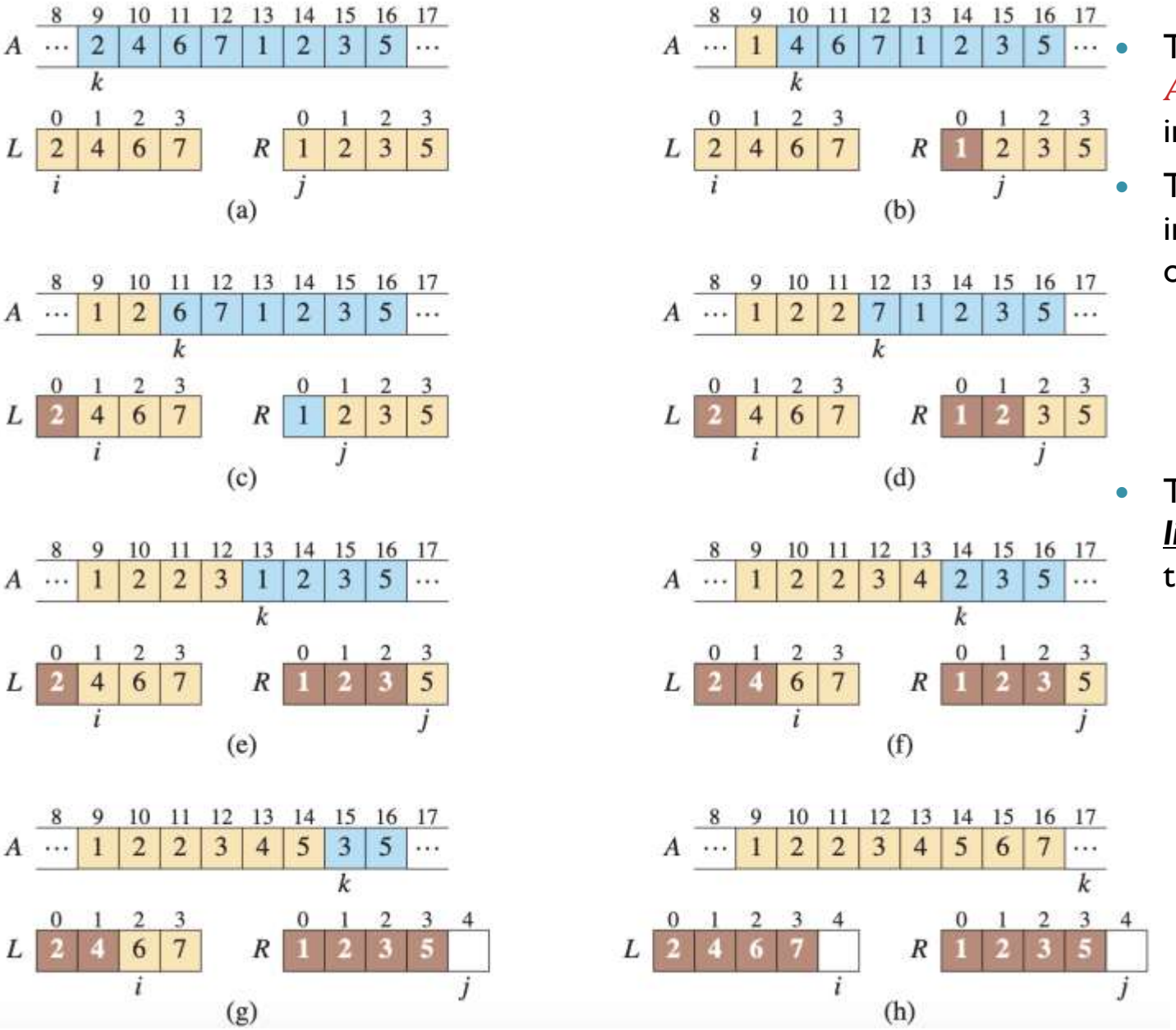
- Invoke the algorithm with $p = 1$ and $r = n$.
- Correctness. Follows easily once we discuss how the MERGE procedure combines the subarrays $A[p{:}\,q]$ and $A[q+1{:}\,r]$ into a sorted array $A[p{:}\,r]$.

Merge: A pictorial depiction



- The two halves of $A[9:16]$ are copied into the arrays L & R.
- The pointers k, i, j are initialized to the start of $A[9:16]$, L & R.

Merge: A pictorial depiction



- The two halves of $A[9:16]$ are copied into the arrays L & R.
- The pointers k, i, j are initialized to the start of $A[9:16]$, L & R.
- Time complexity is ***linear*** in the sizes of the subarrays L and R

Merge: Pseudocode

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Copying the two halves of $A[p:r]$ into the arrays L and R.

Merge: Pseudocode

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Copying the two halves of $A[p:r]$ into the arrays L and R.

Initializing the pointers i, j & k.

Merge: Pseudocode

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Copying the two halves of $A[p:r]$ into the arrays L and R.

Initializing the pointers i, j & k.

Scanning L, R and A using the pointers i, j and k (respectively) and copying the smallest unmerged element into A.

Merge: Pseudocode

```
MERGE(A, p, q, r)
1   n_L = q − p + 1          // length of A[p : q]
2   n_R = r − q              // length of A[q + 1 : r]
3   let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4   for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5       L[i] = A[p + i]
6   for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7       R[j] = A[q + j + 1]
8   i = 0                    // i indexes the smallest remaining element in L
9   j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Copying the two halves of $A[p:r]$ into the arrays L and R.

Initializing the pointers i, j & k.

Scanning L, R and A using the pointers i, j and k (respectively) and copying the smallest unmerged element into A.

Copying the residual part of L or R into A.

Merge: Pseudocode

```
MERGE(A, p, q, r)
1   n_L = q − p + 1          // length of A[p : q]
2   n_R = r − q              // length of A[q + 1 : r]
3   let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4   for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5       L[i] = A[p + i]
6   for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7       R[j] = A[q + j + 1]
8   i = 0                    // i indexes the smallest remaining element in L
9   j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Copying the two halves of $A[p:r]$ into the arrays L and R.

Initializing the pointers i, j & k.

Scanning L, R and A using the pointers i, j and k (respectively) and copying the smallest unmerged element into A.

Copying the residual part of L or R into A.

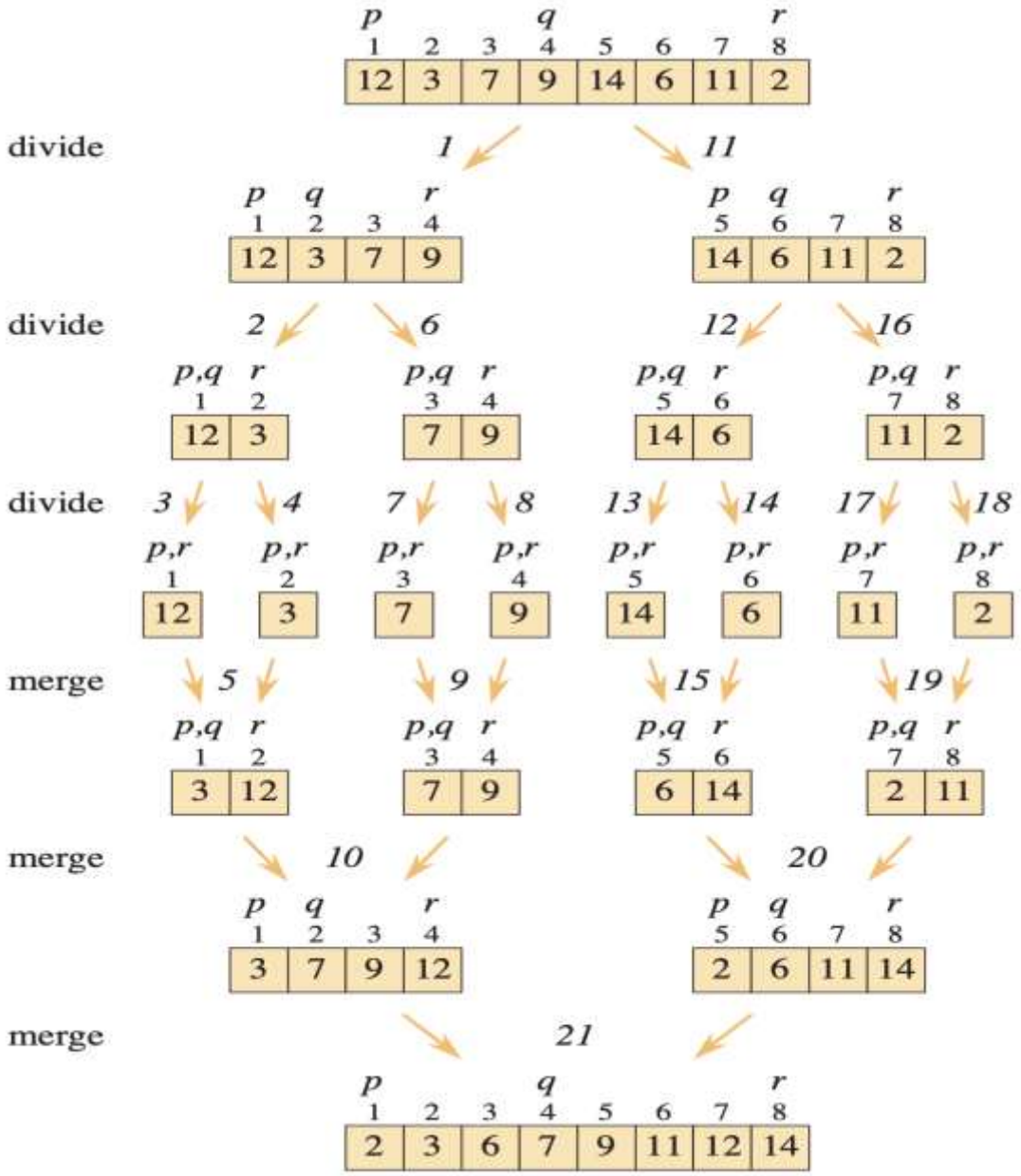
Time complexity is linear in the size of $A[p:r]$.

Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Invoke the algorithm with $p = 1$ and $r = n$.
- Let's see a pictorial depiction of the algorithm in action.

Merge sort: A pictorial depiction



Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Invoke the algorithm with $p = 1$ and $r = n$.
- Worst-case time complexity recurrence relation:

$$T(n) = 2T(n/2) + \Theta(n).$$

Solving the recurrence relation

- $T(n) = 2T(n/2) + \Theta(n)$.

- $\Theta(n)$ means a function that is bounded above and below by $c_1 n$ and $c_2 n$, respectively, for some positive constants c_1 and c_2.

Solving the recurrence relation

- $T(n) = 2T(n/2) + \Theta(n)$

$$\leq 2T(n/2) + c_1 n$$

$$\leq 2^2 T(n/2^2) + c_1 n + c_1 n$$

$$\vdots$$

$$\leq 2^l T(n/2^l) + c_1 n l$$

$$= O(n \log n)$$

Solving the recurrence relation

- $T(n) = 2T(n/2) + \Theta(n)$

$$\geq 2T(n/2) + c_2 n$$

$$\geq 2^2 T(n/2^2) + c_2 n + c_2 n$$

$$\vdots$$

$$\geq 2^l T(n/2^l) + c_2 n l$$

$$= \Omega(n \log n)$$

Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Invoke the algorithm with $p = 1$ and $r = n$.
- Worst-case time complexity of merge sort:

$$T(n) = \Theta(n \log n).$$

Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Invoke the algorithm with $p = 1$ and $r = n$.
- Worst-case time complexity of merge sort:

$$T(n) = \Theta(n \log n).$$

- In fact, the run-time is *always* $\Theta(n \log n)$. Why?

Merge sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                                  // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                           // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)                       // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)                   // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Note. Insertion Sort sorts an array ***in place*** (i.e., only a constant number of array elements are stored outside the input array at any time). Merge sort *doesn't* sort in place.

Matrix multiplication

Matrix multiplication

- Matrix multiplication problem. Given two $n \times n$ matrices $A = (a_{ij})_{i,j \in [n]}$ and $B = (b_{ij})_{i,j \in [n]}$, output the matrix product $C = AB = (c_{ij})_{i,j \in [n]}$.
- The $(i,j)^{th}$ entry of C is $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$.

Matrix multiplication

- Matrix multiplication problem. Given two $n \times n$ matrices $A = (a_{ij})_{i,j \in [n]}$ and $B = (b_{ij})_{i,j \in [n]}$, output the matrix product $C = AB = (c_{ij})_{i,j \in [n]}$.
- The $(i,j)^{th}$ entry of C is $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$.

- Obs. We can compute c_{ij} in $\Theta(n)$ time, assuming that each addition/multiplication takes *constant time*. So, C can be computed in $\Theta(n^3)$ time.

Matrix multiplication: Naïve approach

- Initialize $c_{ij} = 0$ for all $i, j \in [n]$.

```
MATRIX-MULTIPLY(A, B, C, n)
1  for i = 1 to n                    // compute entries in each of n rows
2      for j = 1 to n                // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj   // add in another term of equation (4.1)
```

- Running time. MATRIX-MULTIPLY runs in time $\Theta(n^3)$.

- Question. Can we do better?

Divide-and-conquer approach

- The divide step views each of the $n \times n$ matrices A, B and C as *four* $n/2 \times n/2$ matrices:

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

- Then, we can write the matrix product as:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Divide-and-conquer approach

- The divide step views each of the $n \times n$ matrices A, B and C as *four* $n/2 \times n/2$ matrices:

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

- Then, we can write the matrix product as:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- There are ***eight multiplications*** of $n/2 \times n/2$ matrices in the above expression. Use this to design a recursive algorithm as follows:

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1
 2  // Base case.
 3       c_11 = c_11 + a_11 · b_11
 4       return
 5  // Divide.
 6  partition A, B, and C into n/2 × n/2 submatrices
         A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
         and C_11, C_12, C_21, C_22; respectively
 7  // Conquer.
 8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
 9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10  MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11  MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12  MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13  MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14  MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15  MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1
 2  // Base case.
 3       c_11 = c_11 + a_11 · b_11
 4       return
 5  // Divide.
 6  partition A, B, and C into n/2 × n/2 submatrices
         A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
         and C_11, C_12, C_21, C_22; respectively
 7  // Conquer.
 8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
 9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10  MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11  MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12  MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13  MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14  MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15  MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

Takes $\Theta(1)$ time.

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3       c_11 = c_11 + a_11 · b_11
4       return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
       and C_11, C_12, C_21, C_22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

Takes $\Theta(1)$ time.

Takes $\Theta(1)$ time using *index calculations* to partition matrices.

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3       c_11 = c_11 + a_11 · b_11
4       return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
       and C_11, C_12, C_21, C_22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

Takes $\Theta(1)$ time.

Takes $\Theta(1)$ time using *index calculations* to partition matrices.

Overall running time.

$T(n) = 8T(n/2) + \Theta(1)$

Solving the recurrence relation

- $T(n) = 8T(n/2) + \Theta(1)$

$$\leq 8T(n/2) + c_1$$

$$\leq 8^2T(n/2^2) + 8c_1 + c_1$$

$$\leq 8^3T(n/2^3) + 8^2c_1 + 8c_1 + c_1$$

$$\vdots$$

$$= O(n^3)$$

Solving the recurrence relation

- $T(n) = 8T(n/2) + \Theta(1)$

$$\geq 8T(n/2) + c_2$$

$$\geq 8^2T(n/2^2) + 8c_2 + c_2$$

$$\geq 8^3T(n/2^3) + 8^2c_2 + 8c_2 + c_2$$

$$\vdots$$

$$= \Omega(n^3)$$

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3      c11 = c11 + a11 · b11
4      return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A11, A12, A21, A22; B11, B12, B21, B22;
       and C11, C12, C21, C22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2)
```

Takes $\Theta(1)$ time.

Takes $\Theta(1)$ time using *index calculations* to partition matrices

Overall running time.

$$T(n) = 8T(n/2) + \Theta(1)$$
$$= \Theta(n^3)$$

Alas! We haven't gained anything asymptotically.

Recursive MM: Pseudocode

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3       c_11 = c_11 + a_11 · b_11
4       return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
       and C_11, C_12, C_21, C_22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

Takes $\Theta(1)$ time.

Takes $\Theta(1)$ time using *index calculations* to partition matrices

Overall running time.

$$T(n) = 8T(n/2) + \Theta(1)$$
$$= \Theta(n^3)$$

Alas! We haven't gained anything asymptotically.

Wait! Here comes Strassen..

Strassen's MM algorithm: The idea

- The divide step views each of the $n \times n$ matrices A, B and C as four $n/2 \times n/2$ matrices:

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

- Then, we can write the matrix product as:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- Idea. We **do not need *eight multiplications*** of $n/2 \times n/2$ matrices. ***Seven*** suffices! Here's how…

Strassen's MM algorithm

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

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Then,

- $C_{11} = -P_2 + P_4 + P_5 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_1 - P_3 + P_5 - P_7$

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Then,

- $C_{11} = -P_2 + P_4 + P_5 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_1 - P_3 + P_5 - P_7$

- Strassen (1969). Using **7 multiplications** and 18 additions of $n/2 \times n/2$ matrices, we can compute C.

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Then,

- $C_{11} = -P_2 + P_4 + P_5 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_1 - P_3 + P_5 - P_7$

Seems magical! You needn't memorize these equations

- Strassen (1969). Using **7 multiplications** and 18 additions of $n/2 \times n/2$ matrices, we can compute C.

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Then,

- $C_{11} = -P_2 + P_4 + P_5 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_1 - P_3 + P_5 - P_7$

- Running time. $T(n) = 7T(n/2) + \Theta(n^2)$

Solving the recurrence relation

- $T(n) = 7T(n/2) + \Theta(n^2)$

$$\leq 7T(n/2) + c_1 n^2$$

$$\leq 7^2 T(n/2^2) + 7c_1(n/2)^2 + c_1 n^2$$

$$\leq 7^3 T(n/2^3) + 7^2 c_1 (n/2^2)^2 + 7c_1(n/2)^2 + c_1 n^2$$

$$\vdots$$

$$= O(n^{\log_2 7}) = O(n^{2.81})$$

Solving the recurrence relation

- $T(n) = 7T(n/2) + \Theta(n^2)$

$$\geq 7T(n/2) + c_2 n^2$$

$$\geq 7^2 T(n/2^2) + 7c_2(n/2)^2 + c_2 n^2$$

$$\geq 7^3 T(n/2^3) + 7^2 c_2(n/2^2)^2 + 7c_2(n/2)^2 + c_2 n^2$$

$$\vdots$$

$$= \Omega(n^{\log_2 7}) = \Omega(n^{2.81})$$

Strassen's MM algorithm

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

Suppose,

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

Then,

- $C_{11} = -P_2 + P_4 + P_5 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_1 - P_3 + P_5 - P_7$

- Running time. $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{2.81})$

MM complexity: Current state

- Open question. Can we multiply two $n \times n$ matrices in $\Theta(n^2)$ time?
- *Winograd (1971)* showed that 7 multiplications are *required* for multiplying two 2×2 matrices.

References	M(n)
Strassen (1969)	$O(n^{2.81})$
Schönhage (1981)	$O(n^{2.55})$
Coppersmith & Winograd (1987)	$O(n^{2.376})$
Optimized CW (Stothers, Vassilevska Williams, Le Gall..)	⋮
Alman, Duan, Vassilevska Williams, Xu, Xu, Zhou (2024)	$O(n^{2.37134})$

Recurrence relations

Solving recurrence relations

- The method that we used so far to solve recurrences roughly corresponds to the recursion tree method.
- This method is my personal favorite, as it is *less prone to mistakes*.

Solving recurrence relations

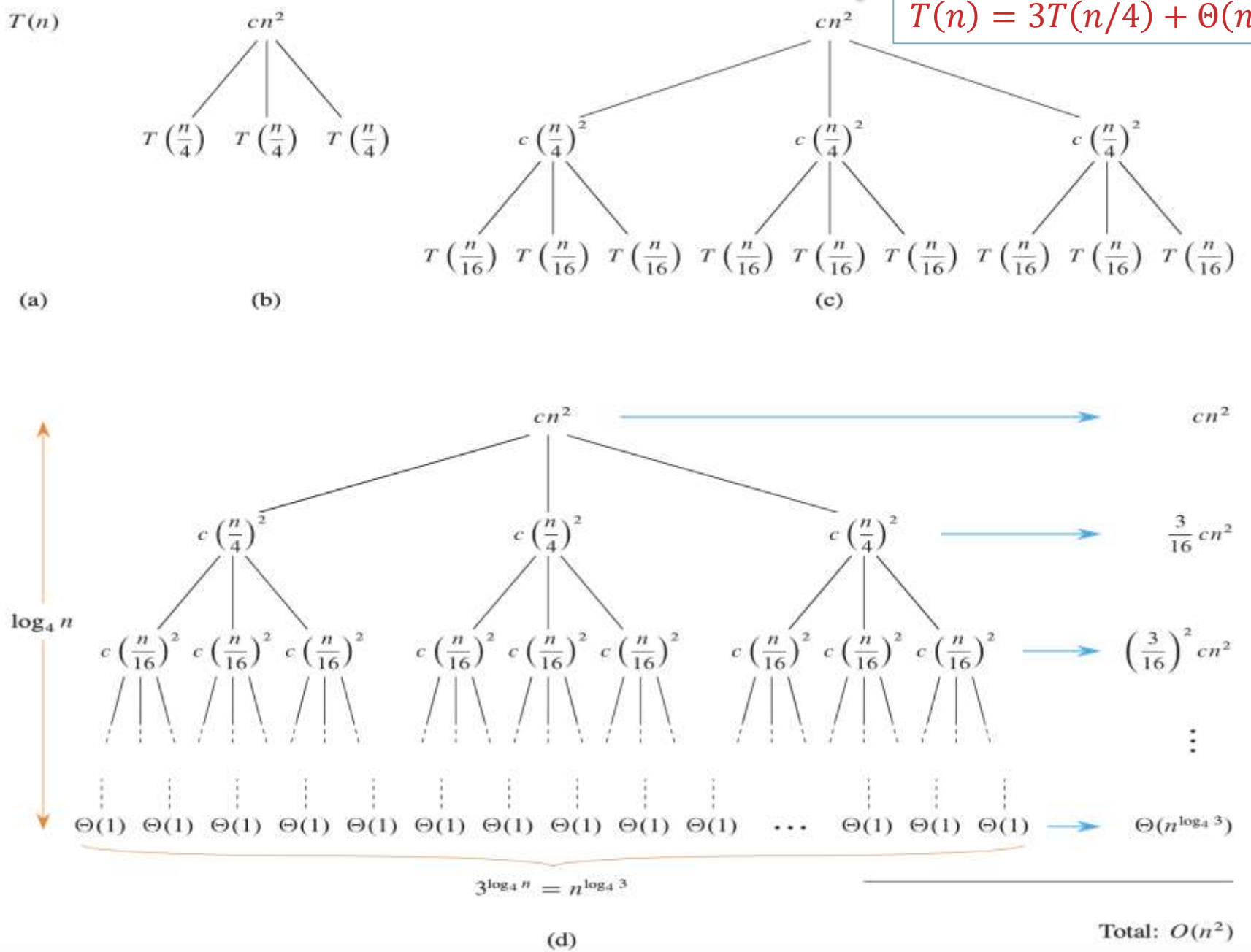
- The method that we used so far to solve recurrences roughly corresponds to the recursion tree method.
- This method is my personal favorite, as it is *less prone to mistakes*.

- Let's see an example…

- Suppose, $T(n) = 3T(n/4) + \Theta(n^2)$.

Recursion tree: An example

$T(n) = 3T(n/4) + \Theta(n^2)$



Solving recurrence relations

- The method that we used so far to solve recurrences roughly corresponds to the recursion tree method.
- This method is my personal favorite, as it is less prone to mistakes.

- Let's see an example…

- Suppose, $T(n) = 3T(n/4) + \Theta(n^2)$
  $= \Theta(n^2)$.

Recursion tree method

- In a recursion tree, *each node represents the cost of a single subproblem* somewhere in the set of recursive function invocations.
- We typically *sum the cost within each level* of the tree to obtain per-level costs.
- Then, we *sum all the per-level costs* to determine the total cost of all levels of the recursion.
- Note. Sometimes, adding up the total cost takes more creativity.

Recursion tree method

- In a recursion tree, *each node represents the cost of a single subproblem* somewhere in the set of recursive function invocations.
- We typically *sum the cost within each level* of the tree to obtain per-level costs.
- Then, we *sum all the per-level costs* to determine the total cost of all levels of the recursion.
- Note. Sometimes, adding up the total cost takes more creativity.

- Next, we display a "***hammer***" for solving recurrences.

The Master theorem

Theorem 4.1 (Master theorem)

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n), \tag{4.17}$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The Master theorem

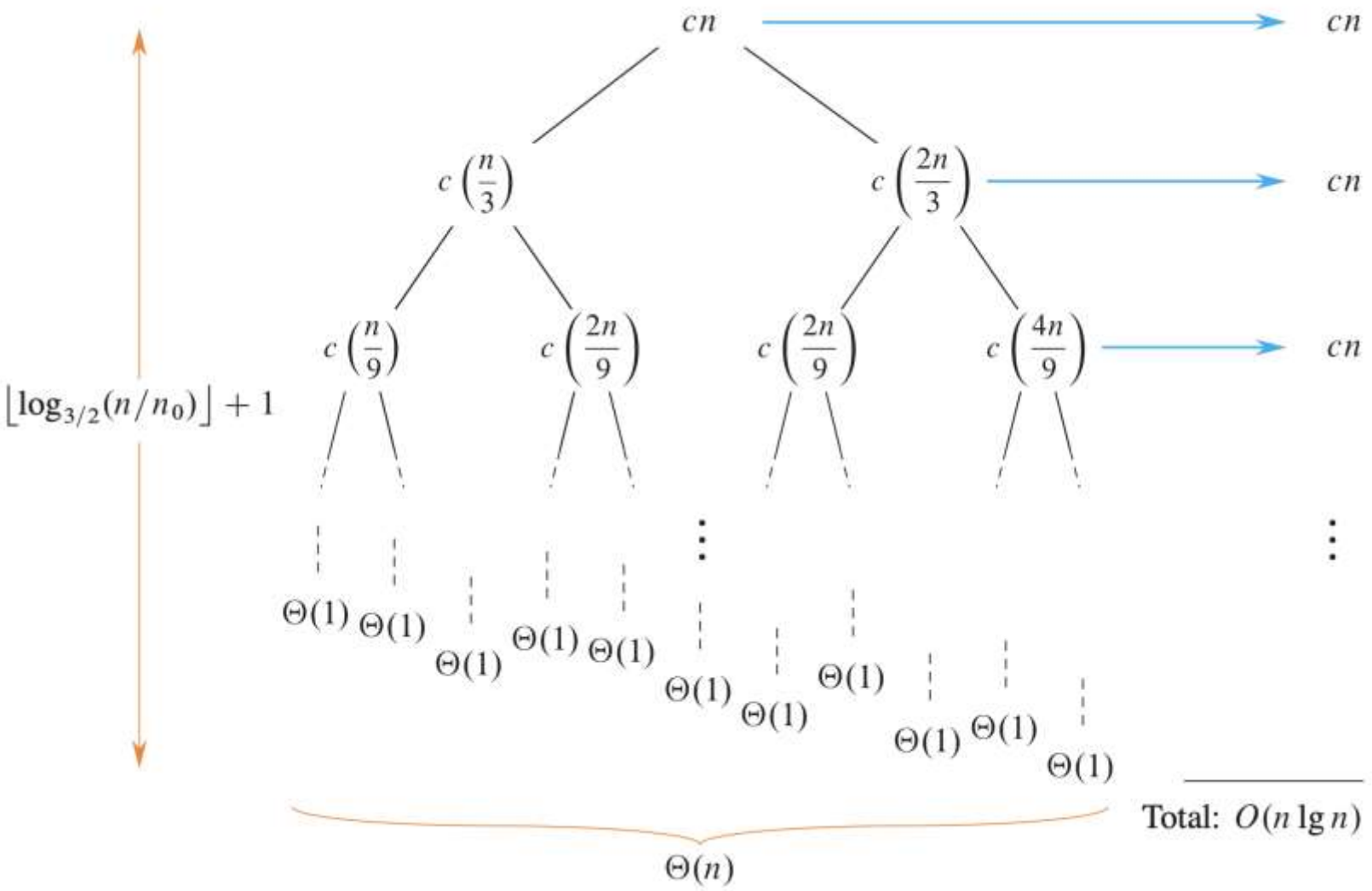
- The Master theorem helps us solve recurrences of the form $T(n) = aT(n/b) + f(n)$ readily, *provided you recall the statement of the theorem accurately*.
- If we don't, then it's better to apply the recursion-tree method. (The proof of the theorem is beyond the scope of this course.)

The Master theorem

- The Master theorem helps us solve recurrences of the form $T(n) = aT(n/b) + f(n)$ readily, *provided you recall the statement of the theorem accurately*.
- If we don't, then it's better to apply the recursion-tree method. (The proof of the theorem is beyond the scope of this course.)
- Drawback of the Master theorem. Does not apply to recurrences that are ***not*** of the form $T(n) = aT(n/b) + f(n)$.
- The recursion tree method is *potentially useful for other kind of recurrences*. Let's see an example.

Recursion tree: An example

- $T(n) = T(n/3) + T(2n/3) + \Theta(n) = \Theta(n \log n)$.



Practice problems

Practice Problem 1

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Practice Problem 2

Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

Practice Problem 3

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Practice Problem 4

Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

Practice Problem 5

Suppose that you have a $\Theta(n^\alpha)$-time algorithm for squaring $n \times n$ matrices, where $\alpha \geq 2$. Show how to use that algorithm to multiply two different $n \times n$ matrices in $\Theta(n^\alpha)$ time.

Practice Problem 6

- Verify the solutions to the following recurrences:

a. $T(n) = T(n-1) + n$ has solution $T(n) = O(n^2)$.

b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

c. $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n \lg n)$.

d. $T(n) = 2T(n/2 + 17) + n$ has solution $T(n) = O(n \lg n)$.

e. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

f. $T(n) = 4T(n/2) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$.

Practice Problem 7

Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following algorithmic recurrences. Justify your answers.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(8n/11) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 4T(n/2) + n^2 \lg n$.

e. $T(n) = 8T(n/3) + n^2$.

f. $T(n) = 7T(n/2) + n^2 \lg n$.

g. $T(n) = 2T(n/4) + \sqrt{n}$.

h. $T(n) = T(n-2) + n^2$.