

Topic 2:
Divide-and-Conquer



Methods for Solving Recurrences

- Divide-and-conquer solves a problem recursively.
- Steps of divide-and-conquer
  - ***Divide*** the problem into a number of subproblems.
  - ***Conquer*** the subproblems by solving them recursively. If the subproblem sizes are small enough, solve them directly.
  - ***Combine*** the solutions of the subproblems into the solution for the original problem.
- Methods for solving recurrences
  - Substitution method
    - Guess a bound and then use mathematical induction to prove our guess correct.
  - Recursion-tree method
    - Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.
  - Master method
    - Provide bounds for recurrences of the form.



Outline

- Merge sort
- Maximum-subarray problem
- Strassen's algorithm for matrix multiplication
- Substitution method
- Recursion-tree method
- Master method



Merge Sort



Divide-and-Conquer Approach

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
  - ***Base case:*** If the subproblems are small enough, just solve them by brute force (暴力法).
- **Combine** the subproblem solutions to give a solution to the original problem.



Merge-Sort

- Because we are dealing with subproblems, we state each subproblem as sorting a subarray *A[p..r]*
  – Initially, *p* = 1 and *r* = *n*, but these values change as we recurse through subproblems. To sort *A[p..r]*
    - **Divide** by splitting into two subarrays *A[p..q]* and *A[q+1..r]*, where q is the halfway point of *A[p..r]*.
    - **Conquer** by recursively sorting the two subarrays *A[p..q]* and *A[q+1..r]*.
    - **Combine** by merging the two sorted subarrays *A[p..q]* and *A[q+1..r]* to produce a single sorted subarray *A[p..r]*.

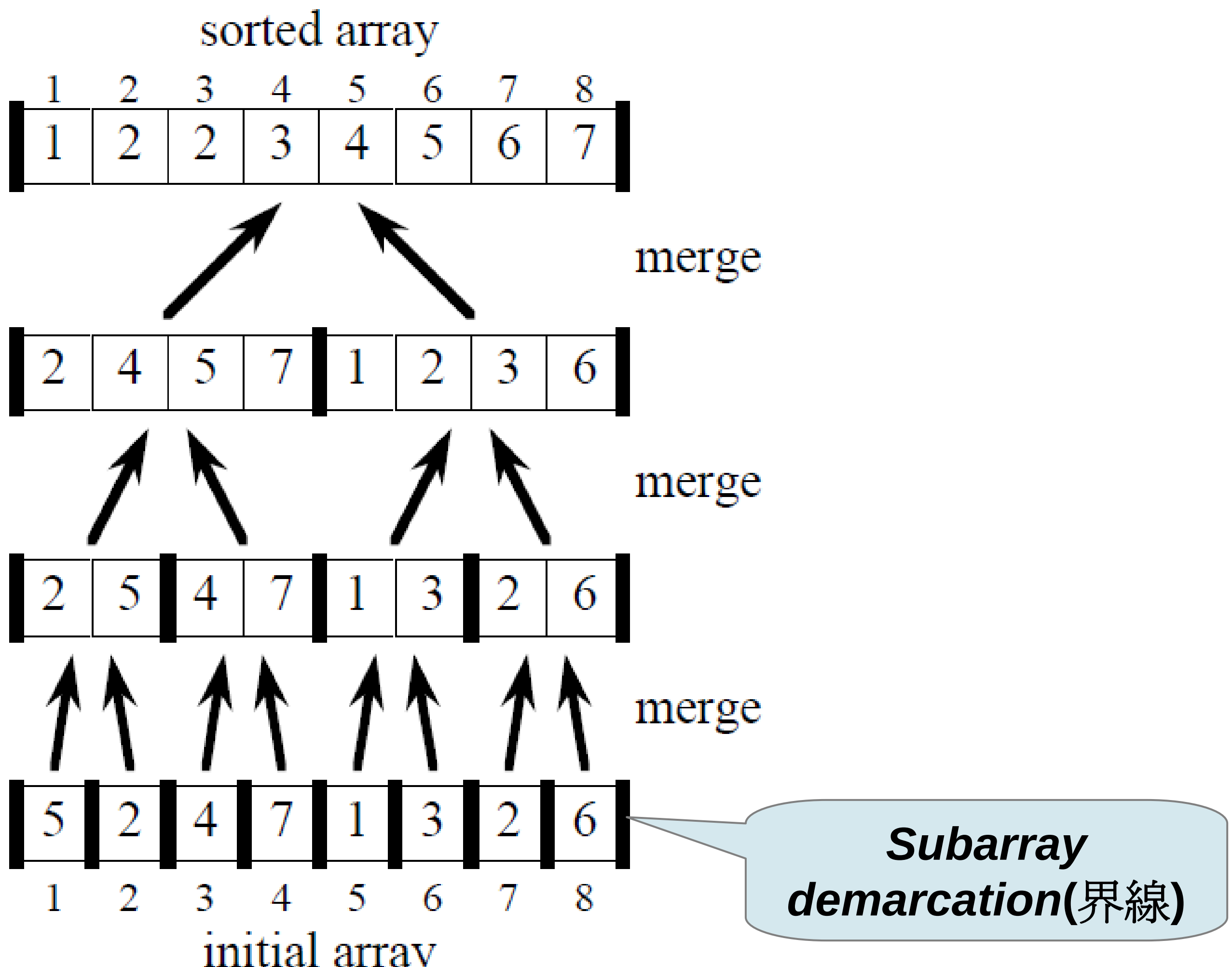
Initial call: MERGE-SORT(A, 1, n)

MERGE-SORT(A, p, r)

if $p < r$ // check for base case
  $q = \lfloor (p + r)/2 \rfloor$ // divide
  MERGE-SORT(A, p, q) // conquer
  MERGE-SORT$(A, q + 1, r)$ // conquer
  MERGE(A, p, q, r) // combine

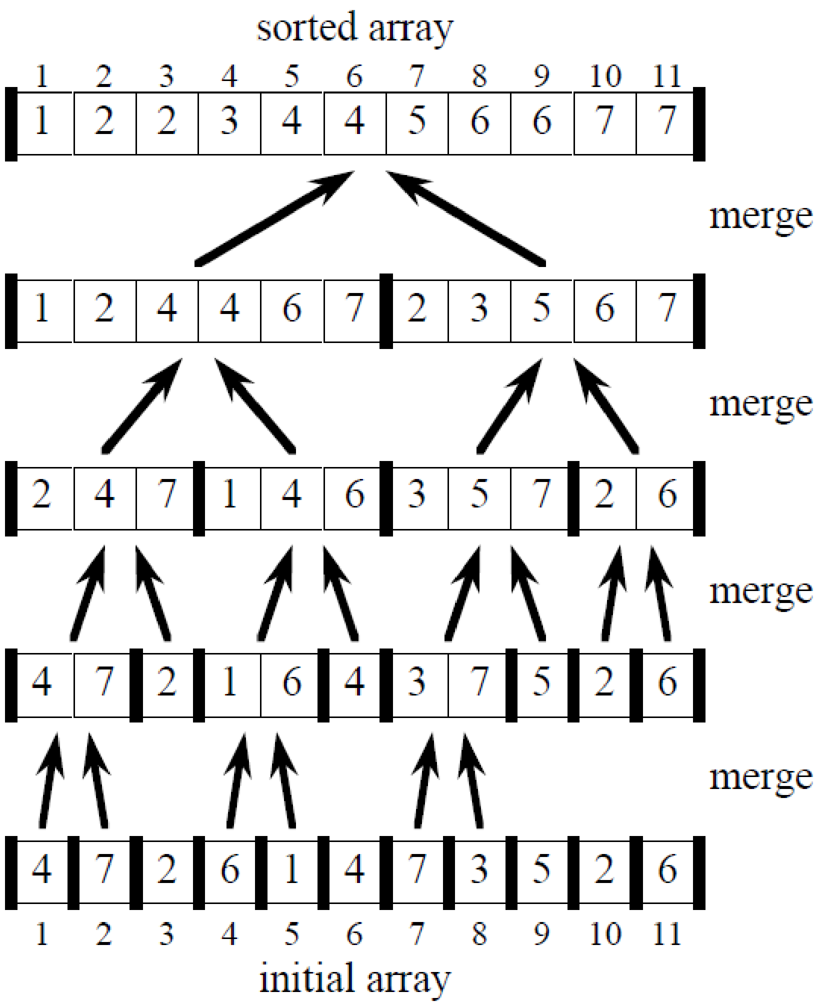
Merge Sort Example

Bottom-up view
for n = 8
(a power of 2)



Merge Sort Example (Cont.)

Bottom-up view for
n = 8
(not a power of 2)





Merging

MERGE(A, p, q, r)

- **INPUT:**
  - Array **A** and indices p, q, r such that
    - $p \leq q < r$.
    - Subarray *A*[*p*..*q*] is sorted and subarray *A*[*q*+1..*r*]. is sorted.
    - By the restrictions on p, q, r, neither subarray is empty.
- **OUTPUT:**
  - The two subarrays are merged into a single sorted subarray in *A*[*p*..*r*].

By adopting ***linear merging***, it takes $\Theta(n)$ time,
where $n = r - p + 1$ = the number of elements being merged.



Merging (Cont.)

- ***Idea behind linear merging:***
  - Think of two piles of cards.
    - Each pile is sorted and placed face-up on a table with the smallest cards on top.
    - We merge these into a single sorted pile, face-down on the table.
    - A basic step:
      - Choose the smaller of the two top cards.
      - Remove it from its pile, thereby exposing a new top card.
      - Place the chosen card face-down onto the output pile.
    - Repeatedly perform basic steps until one input pile is empty.
    - Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

Put on the bottom of each input pile a special ***sentinel*** card. Then We don't actually need to check whether a pile is empty before each basic step.

Merging (Cont.)

Running time:

- The first two **for** loops take $\Theta(n_1 + n_2)$ time.
- The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- Total time: $\Theta(n)$.

MERGE(A, p, q, r)

$n_1 = q - p + 1$

$n_2 = r - q$

let $L[1 \, .. \, n_1 + 1]$ and $R[1 \, .. \, n_2 + 1]$ be new arrays

for $i = 1$ **to** n_1

    $L[i] = A[p + i - 1]$

for $j = 1$ **to** n_2

    $R[j] = A[q + j]$

$L[n_1 + 1] = \infty$

$R[n_2 + 1] = \infty$

Prepare the two sorted arrays to arrays L and R.

Sort and merge arrays L and R back to array A[p..r] (with linear merging)

$i = 1$

$j = 1$

for $k = p$ **to** r

    if $L[i] \le R[j]$

        $A[k] = L[i]$

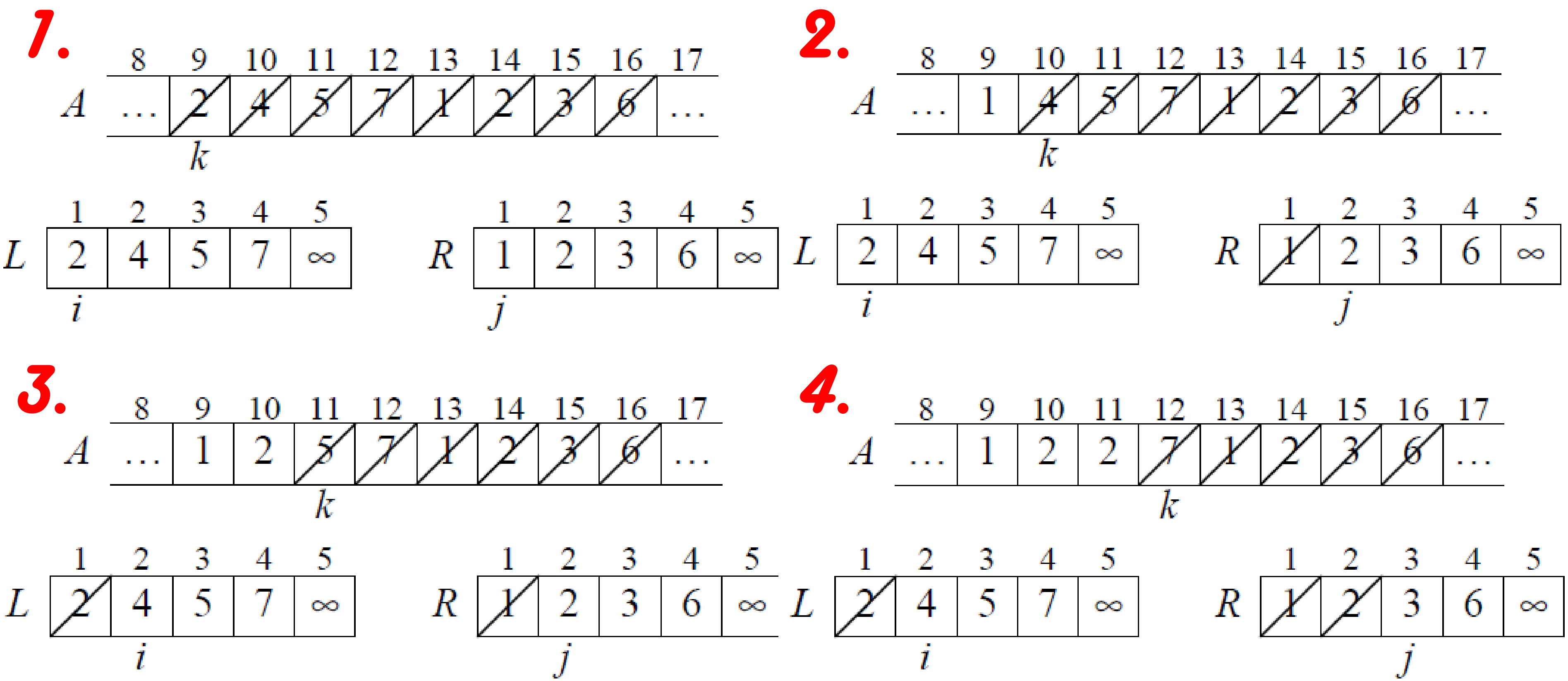
        $i = i + 1$

    else $A[k] = R[j]$

        $j = j + 1$

A Merging Example

A call of MERGE(9, 12, 16)





A Merging Example (Cont.)

5.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	~~1~~	~~2~~	~~3~~	~~6~~	…
						k				

	1	2	3	4	5
L	~~2~~	4	5	7	∞
		i			

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

6.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	~~2~~	~~3~~	~~6~~	…
							k			

	1	2	3	4	5
L	~~2~~	~~4~~	5	7	∞
			i		

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

7.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	~~3~~	~~6~~	…
								k		

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

8.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	~~6~~	…
									k	

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j

9.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	7	…
										k

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	~~7~~	∞
					i

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j



Analyzing Recurrence

- Use a ***recurrence (equation)*** to describe the running time of a divide-and-conquer algorithm.
- Let T(*n*) = running time on a problem of size *n*.

If the problem size is small enough (say, $n \leq c$ for some constant c), we have the base case.
→ ***Brute-force*** solution takes constant time *Q(1)*.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

The time to combine a size-n problem

Suppose that we divide into *a* subproblems, each 1/b the size of the original.
(In merge sort, a = b = 2.)

The time to divide a size-n problem

Analyzing Merge Sort

- Each divide step yields 2 subproblems, both of size exactly $n/2$.
  - The base case occurs when n = 1 $\Rightarrow \Theta(1)$.
  - When $n \geq 2$, time for merge sort steps:
    - **Divide:** Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.
    - **Conquer:** Recursively solve 2 subproblems, each of size n/2 $\Rightarrow 2T(n/2)$.
    - **Combine:** ***MERGE*** on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

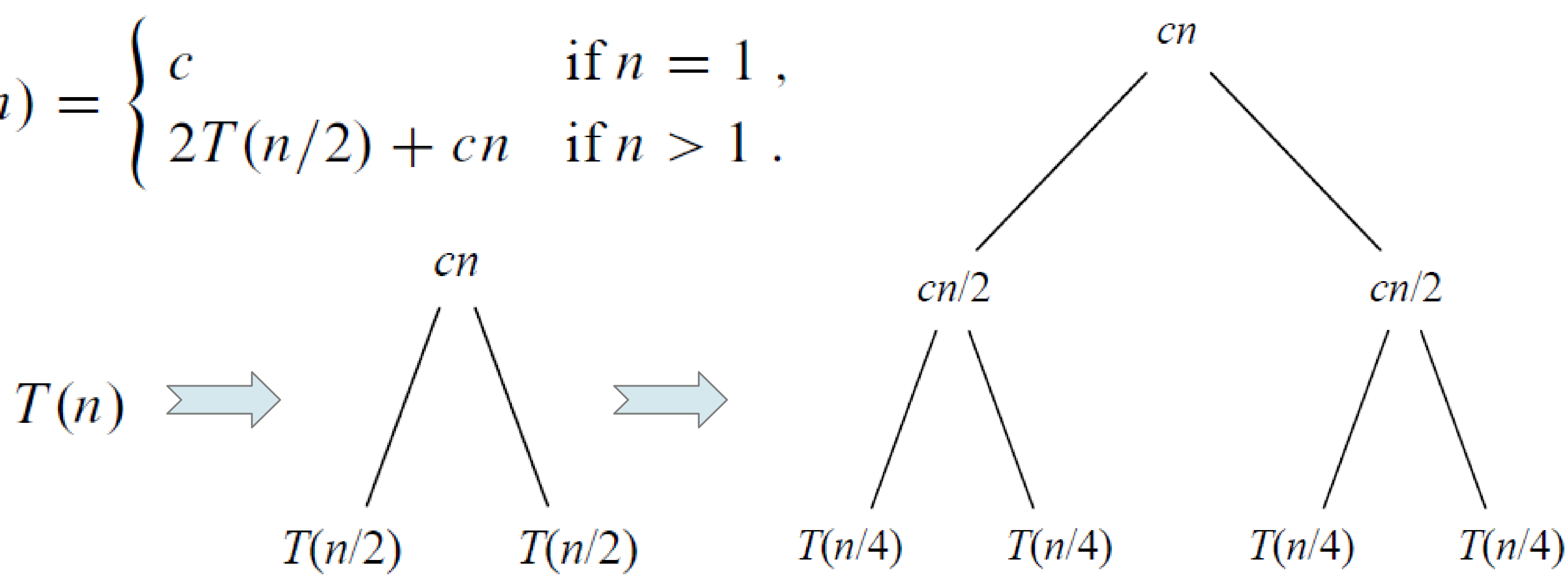
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

$$D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$$

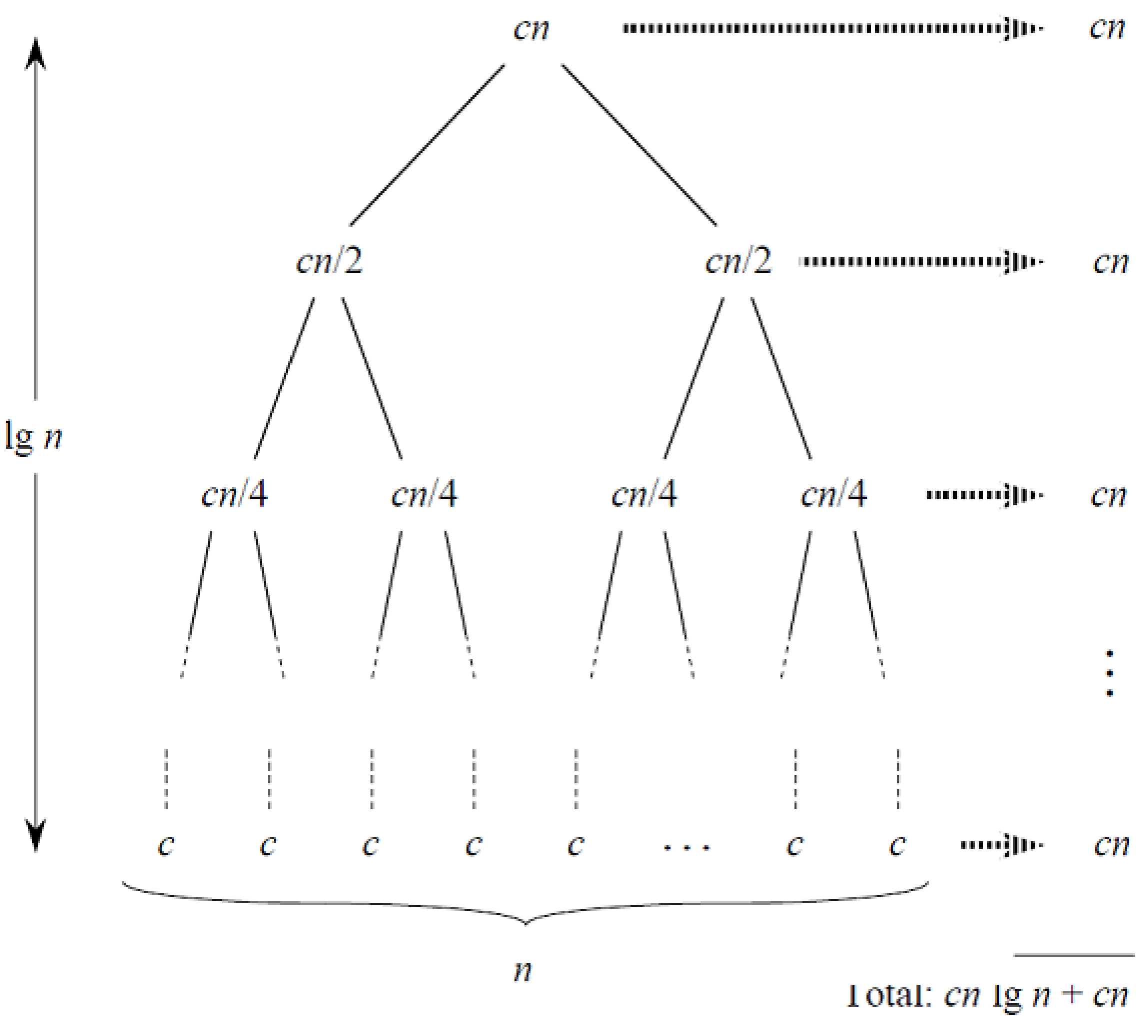
Solving the Merge-Sort Recurrence

- Let *c* be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps.
- We rewrite the recurrence as

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



Solving the Merge-Sort Recurrence (Cont.)



Height: lg n
Levels: lg n + 1
(lg $n = \log_2 n$)
(lg n + 1 = (lg n) + 1)

$T(n) = cn \lg n + cn$
$= \Theta(n \lg n)$



Maximum-Subarray Problem

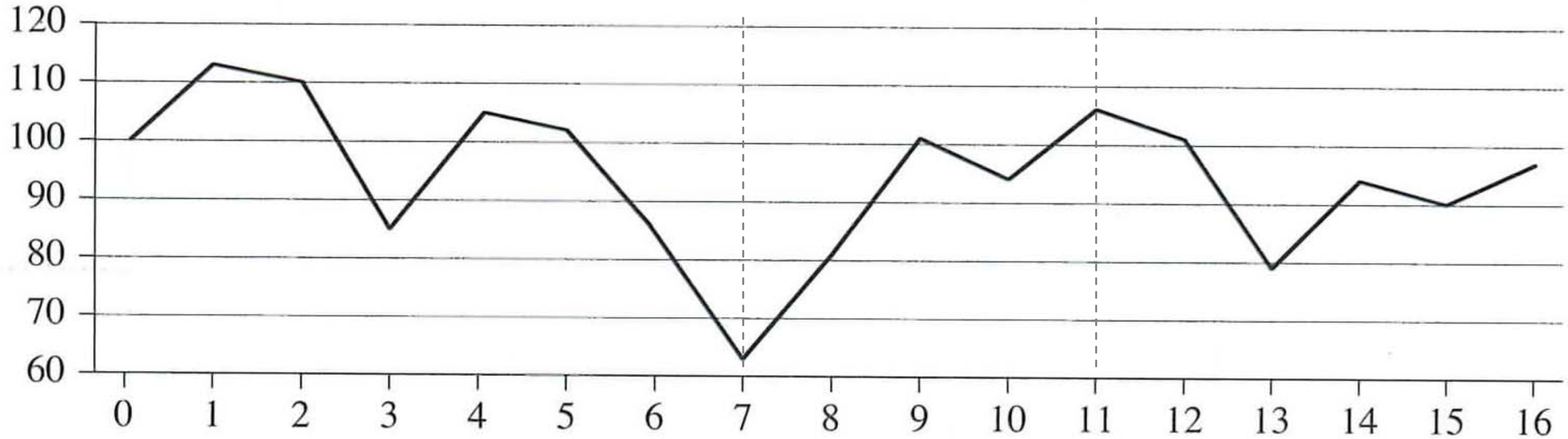
Maximum-Subarray Problem

- **Input:**
  - An array A[1..*n*] of numbers.
  - Assume that some of the numbers are *negative*, because this problem is trivial when all numbers are nonnegative.
- **Output:**
  - Indices *i* and *j* such that A[1..*n*] has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in A[*i*..*j*] .



Scenario

- You have the prices that a stock traded at over a period of *n* consecutive days.
- When should you have bought the stock? When should you have sold the stock?
- Even though it's in retrospect (回顧), you can yell at your stockbroker for not recommending these buy and sell dates. ☺

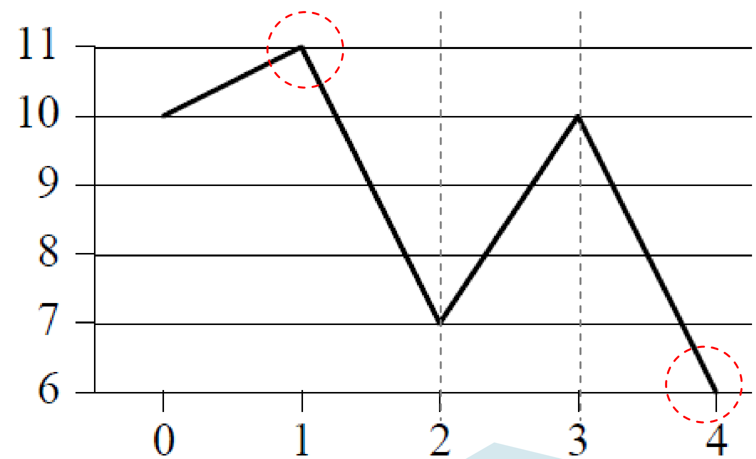


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Maximum profit is A[8..11] = 43 →before day 8 (after day 7) and after day 11

Converting Maximum-Subarray Problem

- Let A[*i*] = (price after day *i*) - (price after day *i-1*)
- If the maximum subarray is A[*i..j*] , then we should
  - Have bought just before day *i* (i.e., just after day *i-1*) and
  - Have sold just after day *j*.
- Why not just “buy low, sell high”?
  - Lowest price might occur *after* the highest price.
  - Maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.



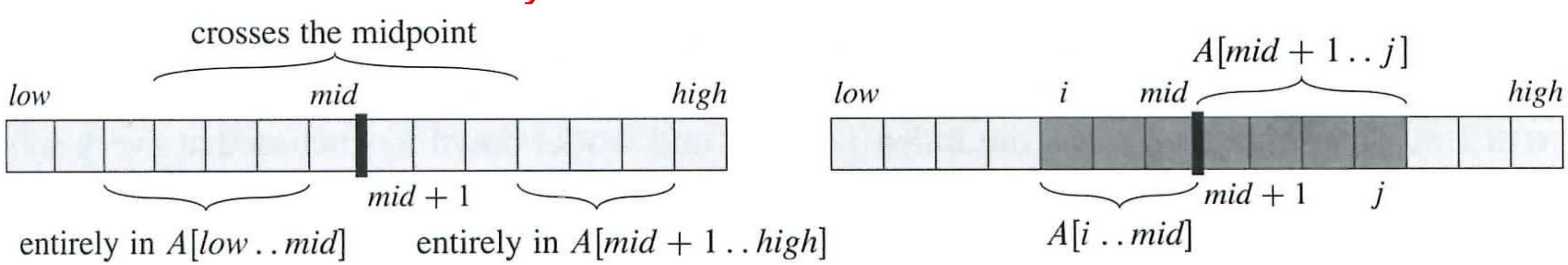
Maximum profit is A[3..3] = 3: before day 3 (after day 2) and after day 3.

- Brute-force solution:

$$\text{check all } \binom{n}{2} = \Theta(n^2) \text{ subarrays}$$

Solving with Divide-and-Conquer

- Divide-and-conquer could solve the maximum-subarray problem in $O(n \lg n)$ time.
- Maximum subarray might not be unique, though its value is.
- ***Subproblem:***
  - Find a maximum subarray of A[*low..high*].
    In original call, *low* = 1, *high* = *n*.
- ***Solving:***
  - **Divide** the subarray into two subarrays of equal size A[*low..mid*] and A[*mid+1..high*].
  - **Conquer** by finding a maximum subarray of A[*low..mid*] and A[*mid+1..high*].
  - **Combine** by finding a maximum subarray that might cross the midpoint or lie on either one subarray.





Maximum Subarray Crossing the Midpoint

- *Not* a smaller instance of the original problem:
  - Any subarray crossing the midpoint A[*mid*] is made of two subarrays A[*i*..mid] and A[mid+1..*j*], where *low≤i≤mid* and *mid <j≤ high*.
  - Find maximum subarrays of the form A[*i*..mid] and A[mid+1..*j*], and then combine them.

This procedure takes Θ(*n*) time.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i..mid].
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid+1..j].
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 the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

From mid to low

From mid to high

Solving Maximum-Subarray Problem

Initial call: $\text{FIND-MAXIMUM-SUBARRAY}(A, 1, n)$

$\text{FIND-MAXIMUM-SUBARRAY}(A, low, high)$

if *high* == *low*

Base case: ***O*(1)**

    return $(low, high, A[low])$ // base case: only one element

else $mid = \lfloor (low + high)/2 \rfloor$

Divide

    $(left\text{-}low, left\text{-}high, left\text{-}sum) =$

        $\text{FIND-MAXIMUM-SUBARRAY}(A, low, mid)$

Search the left subarray

Divide

    $(right\text{-}low, right\text{-}high, right\text{-}sum) =$

        $\text{FIND-MAXIMUM-SUBARRAY}(A, mid + 1, high)$

Search the right subarray

    $(cross\text{-}low, cross\text{-}high, cross\text{-}sum) =$

        $\text{FIND-MAX-CROSSING-SUBARRAY}(A, low, mid, high)$

Crossing the midpoint (combine)

    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$

Left subarray

        return $(left\text{-}low, left\text{-}high, left\text{-}sum)$

    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$

Right subarray

        return $(right\text{-}low, right\text{-}high, right\text{-}sum)$

    else return $(cross\text{-}low, cross\text{-}high, cross\text{-}sum)$

Crossing midpoint

Combine and determine the maximum subarray in A[*low*..*high*]



Analyzing Maximum-Subarray Problem

- ***Base case:***
  - Occurs when *high* equals *low*, so that n = 1. The procedure just returns ⇒ $T(n) = \Theta(1)$.
- ***Recursive case:***
  - Dividing takes $\Theta(1)$ time.
  - Conquering solves 2 subproblems, each on a subarray of n/2 elements ⇒ $2T(n/2)$.
  - Combining consists of
    - Calling FIND-MAX-CROSSING-SUBARRAY ⇒ $\Theta(n)$.
    - A constant number of constant time tests ⇒ $\Theta(1)$.

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \qquad \text{(absorb } \Theta(1) \text{ terms into } \Theta(n)) . \end{aligned}$$

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

Same recurrence as for merge sort

Strassen's Algorithm for Matrix Multiplication



Matrix Multiplication

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$.

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$, i.e.,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

for $i, j = 1, 2, \ldots, n$.

Need to compute n^2 entries of *C*. Each entry is the sum of *n* values.



Obvious Method

SQUARE-MAT-MULT(A, B, n)

let C be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        $c_{ij} = 0$
        for $k = 1$ **to** n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Three nested loops, each iterates n times, and innermost loop body takes constant time $\Rightarrow \Theta(n^3)$



Matrix Multiplication Algorithm

- Assume *n* is a power of 2. Partition each of A, B, C into four *n/2 x n/2* matrices:

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

- Rewrite $C = A \cdot B$ as

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

  - Giving the four equations:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \end{aligned}$$



Matrix Multiplication Algorithm (Cont.)

$\text{REC-MAT-MULT}(A, B)$

let C be a new $n \times n$ matrix

if $n == 1$

$\quad c_{11} = a_{11} \cdot b_{11}$

Base case: ***O*(1)**

else partition A, B, and C into $n/2 \times n/2$ submatrices

$$\begin{aligned}
C_{11} &= \text{REC-MAT-MULT}(A_{11}, B_{11}) + \text{REC-MAT-MULT}(A_{12}, B_{21}) \\
C_{12} &= \text{REC-MAT-MULT}(A_{11}, B_{12}) + \text{REC-MAT-MULT}(A_{12}, B_{22}) \\
C_{21} &= \text{REC-MAT-MULT}(A_{21}, B_{11}) + \text{REC-MAT-MULT}(A_{22}, B_{21}) \\
C_{22} &= \underline{\text{REC-MAT-MULT}(A_{21}, B_{12})} + \text{REC-MAT-MULT}(A_{22}, B_{22})
\end{aligned}$$

return C

$$\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} .
\end{aligned}$$

Eight recursive calls: 8T(n/2)

Four (n/2 x n/2) matrix summation = $n^2/4 \times 4 = n^2$



Analyzing Matrix Multiplication Algorithm

- Let T(*n*) be the time to multiply two *n* x *n* matrices.
- ***Base case:*** n = 1.
  - Perform one scalar multiplication: $\Rightarrow \Theta(1)$.
- ***Recursive case:*** n > 1.
  - Dividing takes
    - $\Theta(1)$ time: using index calculations
    - $\Theta(n^2)$ time: using matrix copying
  - Conquering makes 8 recursive calls, each multiplying *n/2 x n/2* matrices $\Rightarrow 8T(n/2)$.
  - Combining takes $\Theta(n^2)$ time to add *n/2 x n/2* matrices four times (so that it doesn't matter by dividing matrices with index calculation or matrix copying).

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases} \Rightarrow T(n) = \Theta(n^3)$$

Not good enough

$\log_2 8 = 3$



Strassen's Method

- Strassen's algorithm runs in $O(n^{2.81})$ to solve matrix multiplication. How?
  - Perform only 7 recursive multiplications of *n/2 x n/2* matrices, rather than 8.
  - The algorithm:
    - As in the recursive method, partition each of the matrices into four *n/2 x n/2* submatrices. Time: $\Theta(1)$.
    - Create 10 matrices S_1; $S_2 \ldots S_{10}$. Each is *n/2 x n/2* and is the sum or difference of two matrices: Time: $\Theta(n^2)$.
    - Recursively compute 7 matrix products P_1, P_1, …, P_7, each *n/2 x n/2*.
      - Compute *n/2 x n/2* submatrices of *C* by adding and subtracting various combinations of the P_i. Time: $\Theta(n^2)$.

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



Strassen's Method (Cont.)

1.

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

2.

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

3.

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6\,, &&= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}\,,\\
C_{12} &= P_1 + P_2\,, &&= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}\,,\\
C_{21} &= P_3 + P_4\,, &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}\,,\\
C_{22} &= P_5 + P_1 - P_3 - P_7\,. &&= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}\,.
\end{aligned}
$$



Theoretical and Practical Notes

- A method by Coppersmith and Winograd runs in $O(n^{2.376})$ time.
- Practical issues against Strassen's algorithm:
  - Higher constant factor than the obvious $\Theta(n^3)$-time method.
  - Not good for sparse matrices.
    - Many zero rows and columns in sparse matrices
  - Not numerically stable: larger errors accumulate than in the obvious method.
    - Introducing many addition and subtraction operations to the submatrices.
  - Submatrices consume space, especially if copying.



Substitution Method



Substitution Method - Induction

- Two steps of the substitution method:
  - 1. Guess the form of the solution.
  - 2. Use mathematical induction to find constants and show that the solution works.
- ***Example:***

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

- In this example, we have a recurrence with an exact function, rather than asymptotic notation, so that the solution is also exact rather than asymptotic.
- The boundary conditions and the base case should be checked.



Substitution Method – Induction (Cont.)

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

- **_Guess:_** T(n) = Θ(n) = n lg n + n
- **_Induction:_**
  - **_Base:_** $n = 1 \Rightarrow n \lg n + n = 1 = T(1)$
  - **_Inductive step:_**
    - **Inductive hypothesis:** T(k) = k lg k + k, for all k < n
    - Use this inductive hypothesis for T(n/2). Let k = n/2

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\ &= n\lg\frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n\lg n - n + n + n \\ &= n\lg n + n\,. \end{aligned}$$

Induction with Asymptotic Notation

- Technically, with asymptotic notation, we
  - Neglect certain technical details when we state and solve recurrences.
    - A good example of a detail that is often glossed over is the assumption of integer arguments to functions.
  - Ignore boundary conditions.
  - Omit floors and ceilings.
- ***Example***:

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

(We may omit the base case later.)

Induction with Asymptotic Notation (Cont.)

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

- **Guess**: T(*n*) = O(*n* lg *n*) ≤ *cn* lg *n*
- ***Induction:***
  - ***Base:***
    - n = 1 ⇒ T(1)=1, Guess: T(1) = c × 1 × lg 1 = c × 1 × 0 = 0 (→←: conflict)
    - n = 2 ⇒ T(2)=2T(1)+2=4, Guess: T(2) = c × 2 × lg 2 = c × 2 × 1 = 2c (It holds when c ≥ 2 and n=2)
  - ***Inductive hypothesis:*** T(*k*) = c*k* lg *k*, for all *k* < *n*
    - Use this inductive hypothesis for T(*n*/2). Let k = ⌊*n*/2⌋

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\le 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor) + n \\ &\le cn \lg \frac{n}{2} + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n = cn \lg n + (1-c)n \\ &\le cn \lg n \qquad \text{(if } c \ge 1) \end{aligned}$$

T(*n*) = O(*n* lg *n*)
when c ≥ 2 and n ≥ 2

Avoiding Pitfalls

- ***Example:***

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

- ***Guess***:

$$T(n) = O(n) \Rightarrow T(n) \le cn$$

- ***Induction***:

$$T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n = O(n)$$ →Wrong

$$T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n \not\le cn = O(n)$$

→c should be a positive integer, so there is no such c to let $cn + n \le cn$



Subtleties

- Consider the recurrence:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

- ***Wrong guess:***
  - ***Guess***:

$$T(n) = O(n) \quad \Rightarrow T(n) \le cn$$

  - ***Induction***:

$$T(n) \le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \le cn + 1 \not\le cn$$

- ***A proper guess:***
  - ***Guess***:

$$T(n) \le cn - b$$

  - ***Induction***: $T(n) \le (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1$

$$\le cn - 2b + 1 \le cn - b$$ (Choose b≥1)

Changing Variables

- ***Example:***

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

- ***Ignore rounding***
  → This does not affect the derived time complexity

$$\boxed{T(n) = 2T(\sqrt{n}) + \lg n}$$

- **Let** *m* = lg *n* ⇒ $2^m = n$

$$T(n) = T(2^m) = 2T(2^{m/2}) + m$$

- **Let** *S(m) = T(2^m)* ⇒

$$S(m) = 2S(m/2) + m$$

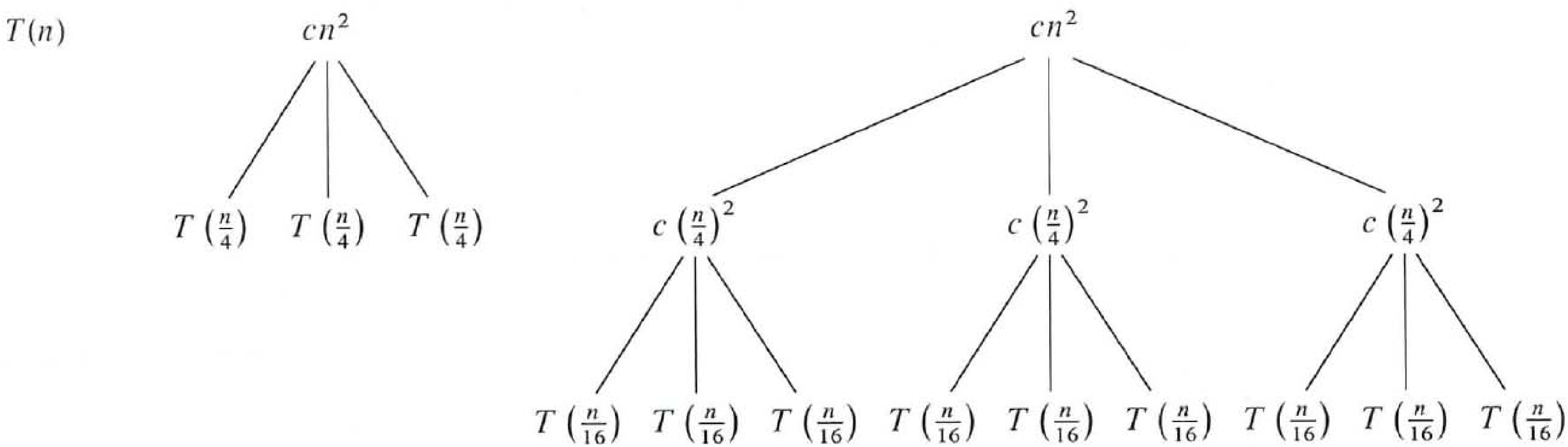
⇒O(*m* lg *m*)
⇒ T(n) = T(2^m) = S(m)
= O(m lg m) = O(lg*n* lglg*n*)

Recursion-Tree Method

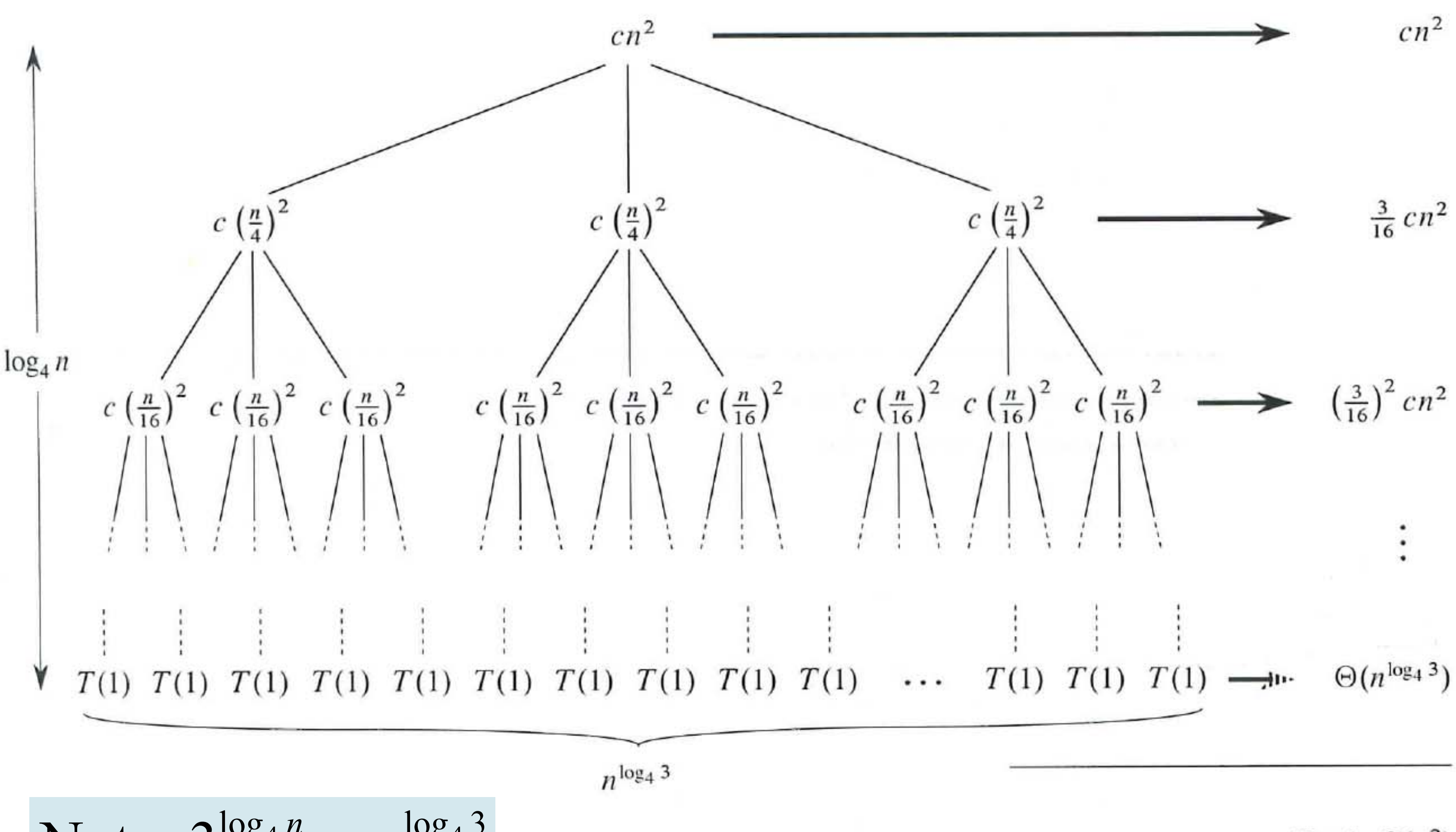
Recursion-Tree Method

- Example: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$
  - Suppose n is a power of 2
  - $\Theta(n^2) = cn^2$



(a) (b) (c)

Recursion-Tree Method (Cont.)



$$\text{Note}: 3^{\log_4 n} = n^{\log_4 3}$$

(d)



Recursion-Tree Method (Cont.)

- Cost of $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$ is as follows, where $\Theta(n^2) = cn^2$

$$T(n)=cn^2+\frac{3}{16}cn^2+\left(\frac{3}{16}\right)^2 cn^2+\ldots+\left(\frac{3}{16}\right)^{\log_4 n-1} cn^2+\Theta(n^{\log_4 3})$$

$$=\sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta\left(n^{\log_4 3}\right)$$

Geometric series with common ratio: 3/16

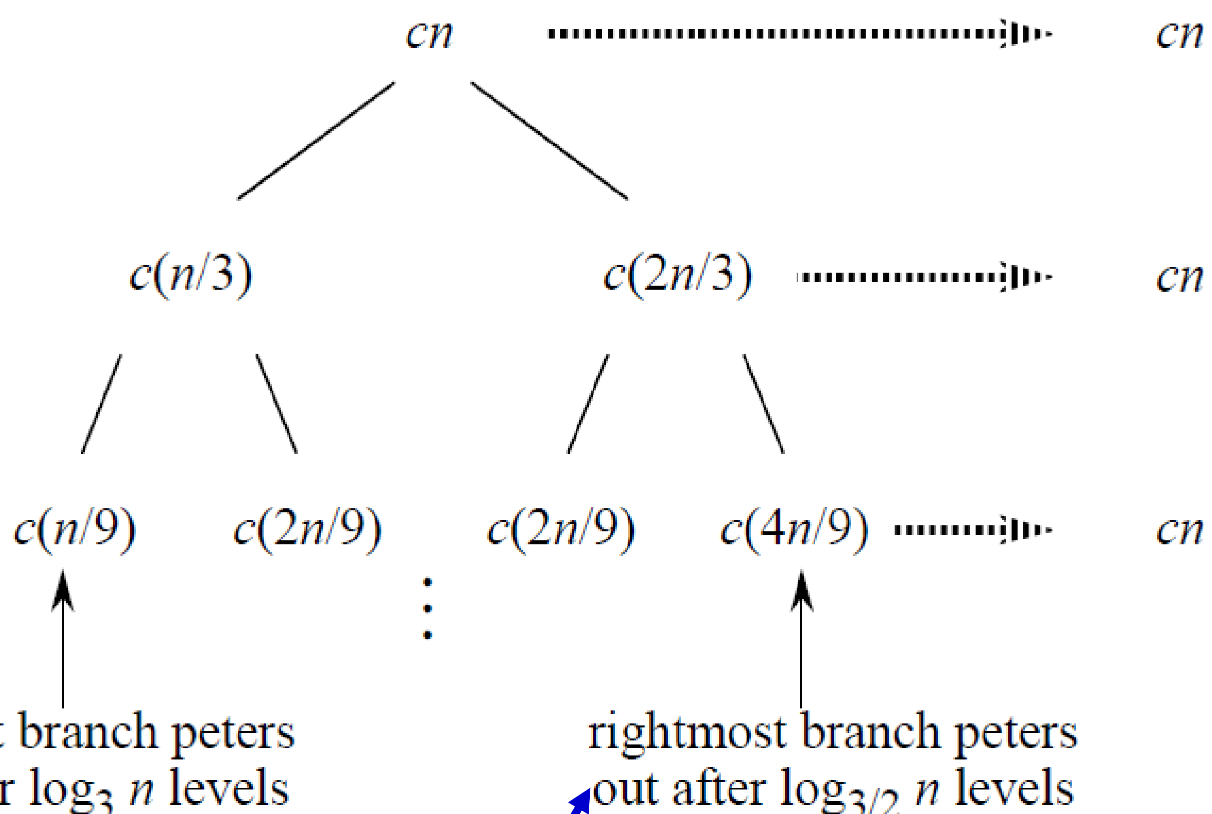
$$=\frac{(3/16)^{\log_4 n}-1}{(3/16)-1}cn^2+\Theta(n^{\log_4 3})$$

$$\le\frac{1}{1-(3/16)}cn^2+\Theta(n^{\log_4 3})$$

$$=\frac{16}{13}cn^2+\Theta(n^{\log_4 3})=O(n^2)$$

Another Example

$$T(n) = T(n/3) + T(2n/3) + O(n) \le T(n/3) + T(2n/3) + cn$$



Upper bound guess:

$$T(n) \le d\, n \log_{3/2} n = O(n \lg n) \text{ for some positive constant } d.$$



Another Example (Cont.)

Guess: $T(n) \le dn\lg n$.

Substitution:

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n \underline{- d(n/3)\lg 3)} \\
&\qquad + (d(2n/3)\lg n \underline{- d(2n/3)\lg(3/2))} + cn \\
&= dn\lg n \underline{- d((n/3)\lg 3 + (2n/3)\lg(3/2))} + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n \qquad \text{if } -dn(\lg 3 - 2/3) + cn \le 0, \\
&\qquad\qquad\qquad\qquad\qquad d \ge \frac{c}{\lg 3 - 2/3}.
\end{aligned}
$$

Therefore, $T(n) = O(n\lg n)$.

Master Method

Master Theorem

$$T(n) = aT(n/b) + f(n)\ ,$$
where $a \geq 1$, $b > 1$, and $f(n) > 0$.

Recurrence form

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

$f(n)$ is polynomially smaller than $n^{\log_b a}$. n^{ε}

Intuitively: cost is dominated by leaves.

Solution: $T(n) = \Theta(n^{\log_b a})$.

The number of leave nodes

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

$f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and

$f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

$f(n)$ is polynomially greater than $n^{\log_b a}$. n^{ε}

Intuitively: cost is dominated by root.

Solution: $T(n) = \Theta(f(n))$.

Using Master Theorem

$T(n) = 5T(n/2) + \Theta(n^2)$

$n^{\log_2 5}$ vs. n^2

Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$,

use Case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$

$T(n) = 27T(n/3) + \Theta(n^3 \lg n)$

$n^{\log_3 27} = n^3$ vs. $n^3 \lg n$

Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

$T(n) = 5T(n/2) + \Theta(n^3)$

$n^{\log_2 5}$ vs. n^3

Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$

$af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$

Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

Cannot use the master method.

Not polynomial larger or smaller

$T(n) = 27T(n/3) + \Theta(n^3/\lg n)$

$n^{\log_3 27} = n^3$ vs. $n^3/\lg n = n^3 \lg^{-1} n \ne \Theta(n^3 \lg^k n)$ for any $k \ge 0$.



Project 2

- Use C language to implement the merge sort with divide-and-conquer.
  - Use *fscanf()* to get integers from the input file.
    - The first integer indicate the number of input integers in this file.
    - E.g., "***3 34 45 67***" means there are three integers that are 34, 45, and 67.
  - Use malloc() to allocate memory space for the input.
  - Sort the input integers and output the sorted integers in the ***monotonically increasing*** order on the screen.
- Deadline: 24:00, 2010.09.27
  - Email the .c or .cpp program to me: johnsonchang@ntut.edu.tw
  - Email title: Algo_P2_學號_姓名



Project 3

- Use C language to implement the maximum-subarray problem with divide-and-conquer.
  - The input file should be retrieved through ***argv[1]*** of main() function.
  - Use *fscanf()* to get integers from the input file.
    - The first integer indicate the number of input integers in this file.
    - E.g., "***4 1 4 3 -4***" means there are four changes that are 1, 4, 3 and -4.
  - Find and output the maximal interval and the maximal revenue.
    - .E.g., 1..3, 8
- Deadline: 24:00, 2010.10.04
  - Email the .c or .cpp program to me: johnsonchang@ntut.edu.tw
  - Email title: Algo_P3_學號_姓名