GALGOTIAS UNIVERSITY

School of Computing Science and Engineering

Program: BCA

Course Code: BCAS2104

Course Name: Introduction to Algorithm Analysis & Design (ADA)

Course Outcomes :

CO1	To know about fundamentals of algorithm.
CO2	To understand the concepts of advanced data structure.
CO3	Apply algorithms and design techniques to solve problems.
CO4	Apply Graph algorithm to find a shortest path.
CO5	To learn about algebraic computation and string matching.
CO6	To understand the latest research trends in Algorithm Design.

Course Prerequisites

Data and File Structures. Knowledge and experience of programming in a high level language like C, C++

Syllabus

Unit I: Introduction to Algorithms: **8 lecture hours**

Introduction to Algorithms & Analysis- Design of Algorithms, Growth of function, Complexity of Algorithms, Asymptotic Notations, Recurrences.

Sorting: Insertion Sort, Quick Sort, Merge Sort

Unit II: Advance Data Structure: **6 lecture hours**

Advanced Data Structure: Binary Search Trees, Red Black Trees, B-Tree

Unit III: Advance Design and Analysis Techniques: **7 lecture hours**

Advanced Design and Analysis Techniques: Dynamic programming-Matrix chain multiplication, Longest common sequence and Knapsack problem, Greedy Algorithm-Huffman Coding, and Knapsack problem

Unit IV: Graph Algorithms **8 lecture hours**

Graph Algorithms: Elementary Graph Algorithms, Breadth First Search, Depth First Search, Minimum Spanning Tree, Kruskal's Algorithms, Prim's Algorithms, Single Source Shortest Path

Unit V: Special Topics in AAD **5 lecture hours**

String Matching, Introduction of NP-Hard and NP-Completeness, Matrix Operations

Unit VI: Research **6 lecture hours**

Research Topics : Approximation Algorithms : The Traveling Salesman Problem.

Discussion of some latest papers published in IEEE transactions and ACM transactions, Web of Science and SCOPUS indexed journals as well as high impact factor conferences as well as symposiums.

Recommended Books

Text books

- T. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein – Introduction to Algorithms – PHI – 2nd Edition, 2005.

Reference Book

- Knuth E. Donald, Art of Computer Programming Sorting and Searching Vol3, Second Edition, Pearson Education.
- Brassard Bratley, “Fundamental of Algorithms”, PHI
- A V Aho etal, “The Design and analysis of Algorithms”, Pearson Education
- Adam Drozdek, “Data Structures and Algorithms in C++”, Thomson Asia

Additional online materials

What is an algorithm?

- An algorithm is a finite set of instructions for solving a problem.
- An algorithm step by step process.
- Algorithms are the ideas behind computer programs.

- An algorithm has to solve a general, specified problem. An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

Algorithms

- Properties of algorithms: *(all algorithms must satisfy the following properties)*
- **Input:** Zero or more quantities are externally supplied.
- **Output:** At least one quantity is produced(solution).
- **Definiteness**: Each instruction is clear and unambiguous
- **Finiteness:** Algorithm must be terminate after a finite number of steps.
- **Effectiveness**: It also must be feasible(desire output).

Growth of Function

- Asymptotic notation (O, Θ, Ω)
  - Big-oh O
  - Theta Θ
  - Omega Ω
- Informal definitions:
  - Given a complexity function f(n),
  - Ω(f(n)) is the set of complexity functions that are *lower bounds* on f(n)
  - O(f(n)) is the set of complexity functions that are *upper bounds* on f(n)
  - Θ(f(n)) is the set of complexity functions that, given the correct constants, "correctly" describes f(n)

Asymptotic Notations

- A way to describe behavior of functions in the limit
  - How we indicate running times of algorithms
  - Describe the running time of an algorithm as n grows to $\propto$
- O notation: asymptotic "less than and equal": f(n) "≤" cg(n)
- Ω notation: asymptotic "greater than and equal": f(n) "≥" cg(n)
- Θ notation: asymptotic "equality": f(n) "=" cg(n)

$$F(n) = n^2 + n + 1$$

$$G(n) = O(n^2)$$

Analysis of algorithm

- Analysis of algorithm or performance analysis refers to the task of determine how much computing (CPU) time and storage (RAM) time requires.
  - Time complexity
    - The time complexity of an algorithm is the amount of computer time it needs to run to completion.
  - Space complexity
    - The space complexity of an algorithm is the amount of memory it needs to run to completion.

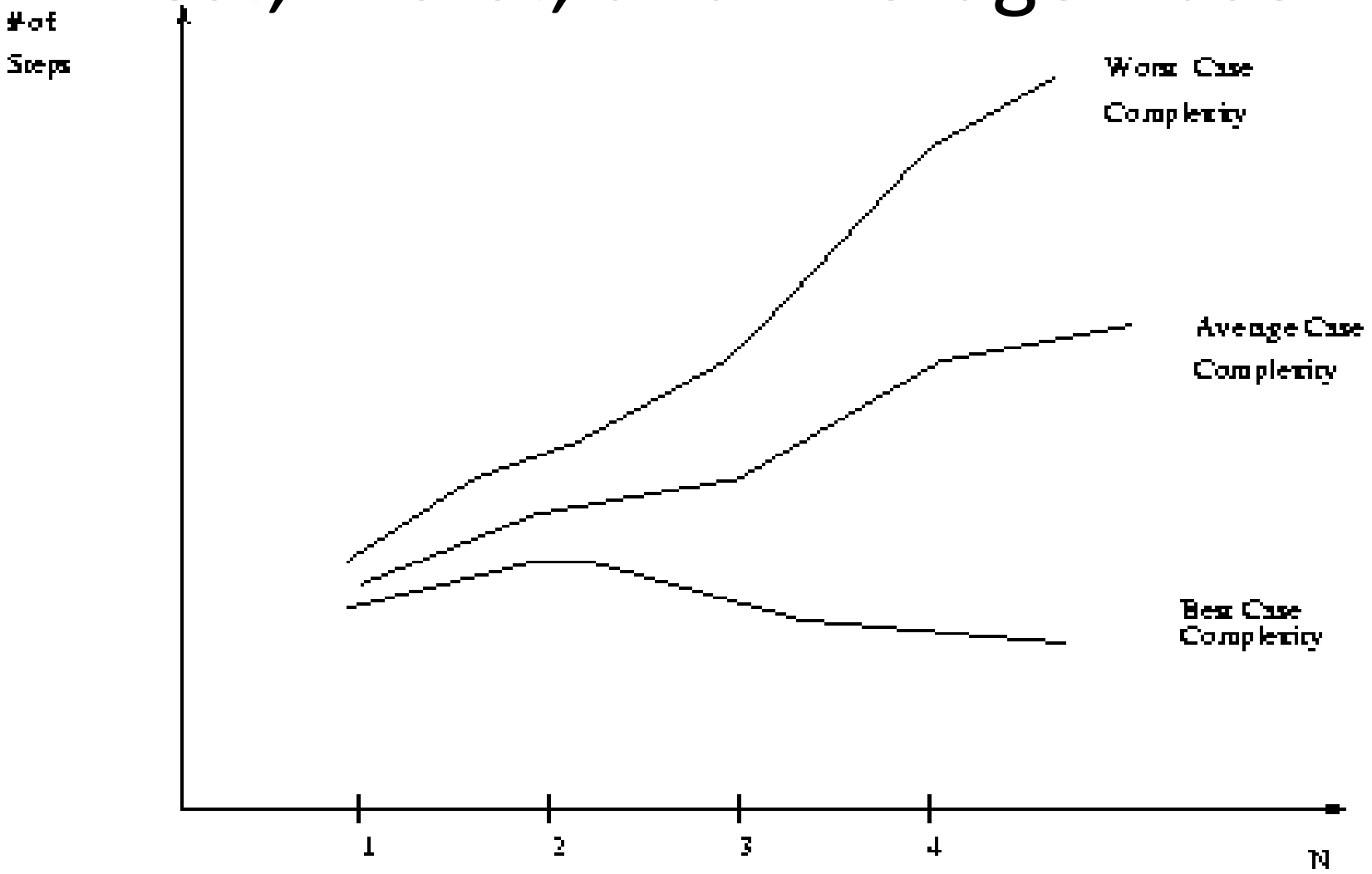
Continue

- Input Size: Size of problem say n
  - The best notation for input size depends on the problem being studied.
- Running Time: T(n)
  - The running time of an algorithm on a particular input is the number of primitive operations or steps executed.

Best, Worst, and Average-Case

- The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.
- The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.
- The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size *n*.
- Each of these complexities defines a numerical function - time vs. size!

Best, Worst, and Average-Case



Asymptotic Notations

- A way to describe behavior of functions in the limit
  - How we indicate running times of algorithms
  - Describe the running time of an algorithm as n grows to $\propto$
- O notation: asymptotic "less than and equal": f(n) "≤" cg(n)
- Ω notation: asymptotic "greater than and equal": f(n) "≥" cg(n)
- Θ notation: asymptotic "equality": f(n) "=" cg(n)

$$F(n) = n^2 + n + 1$$

$$G(n) = O(n^2)$$

Recurrence relation

- A recurrence relation is an equation which is defined in terms of itself.
- There are three type of recurrence solutions :

Recurrence Relations

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- **Solution Methods** (Chapter 4)
  - Substitution Method.
  - Recursion-tree Method.
  - Master Method.
- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.
  - **Ex:** Divide and Conquer.

$$T(n) = \Theta(1) \quad \text{if } n \leq c$$

$$T(n) = a\,T(n/b) + D(n) + C(n) \quad \text{otherwise}$$

Substitution Method

- **Guess** the form of the solution, then **use mathematical induction** to show it correct.
  - Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values – hence, the name.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

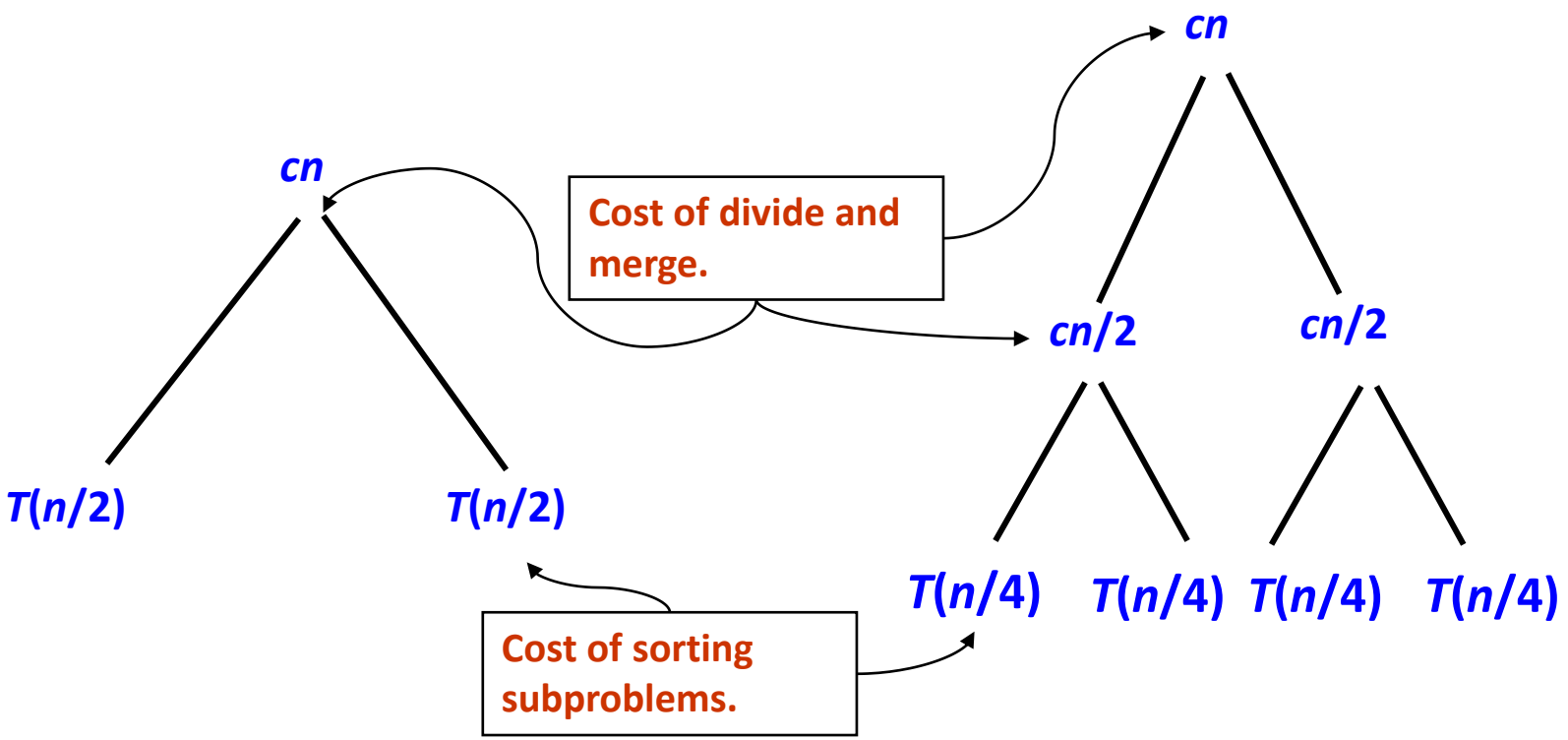
1. Total cost of each sub problems are equal

- Like merge sort recurrence relation
- $T(n)=2T(n/2)+ cn$ **if** $n>1$

Recursion Tree for Merge Sort

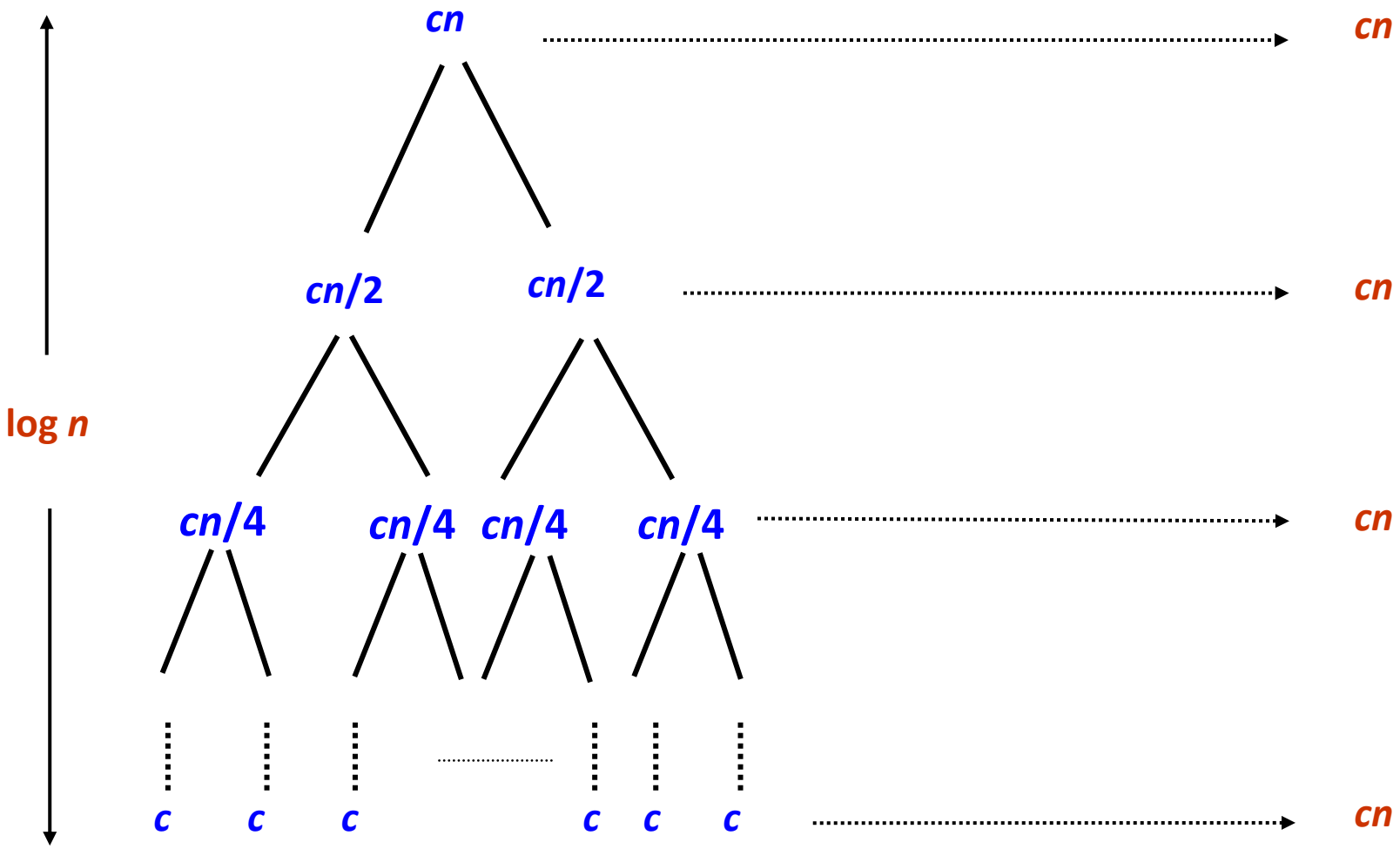
For the original problem, we have a cost of *cn*, plus two subproblems each of size ($n/2$) and running time $T(n/2)$.

Each of the size $n/2$ problems has a cost of $cn/2$ plus two subproblems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



Total : $cn\log n + cn$

Recursion Tree

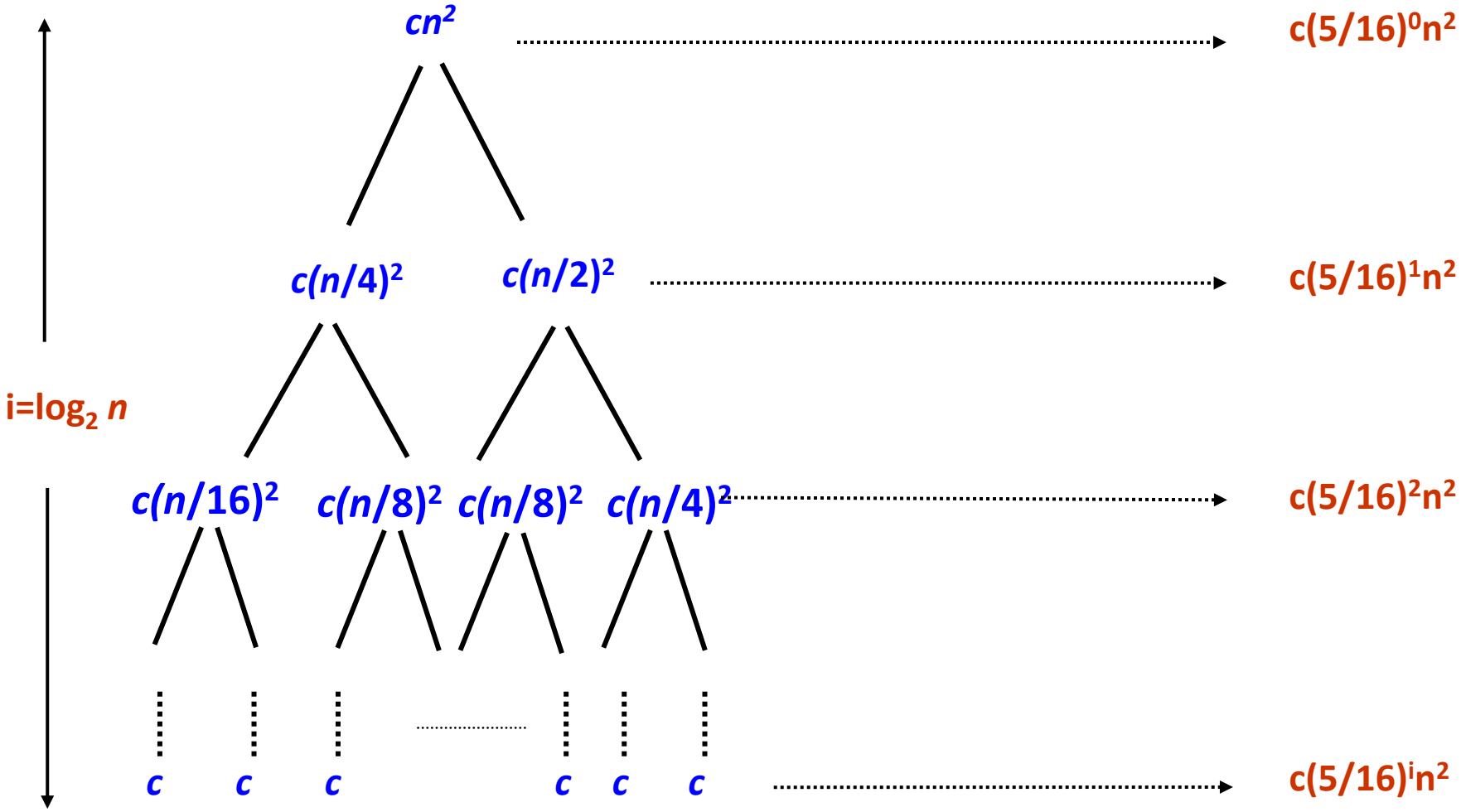
- At i^{th} level no. of nodes 2^i,
- Sub-problem size at level i is: $n/2^i$
- Sub-problem size hits 1 when $1 = n/2^i \Rightarrow n = 2^i$
  - Taking log, $i = \log_2 n$ (Height of tree)
- Total cost = cost of each level × Height of tree
- $T(n) = cn (\log n + 1)$
- $T(n) = cn\log n + cn$
  - Ignore low-order term ***n*** and constant coefficient ***c***
- $T(n) = \Theta(n \log n)$,

2. Total cost of each sub-problems are not equal and size of sub-problems are also not equal.

- $T(n)=T(n/2)+T(n/4)+n^2$

Recursion Tree

Continue expanding until the problem size reduces to 1.



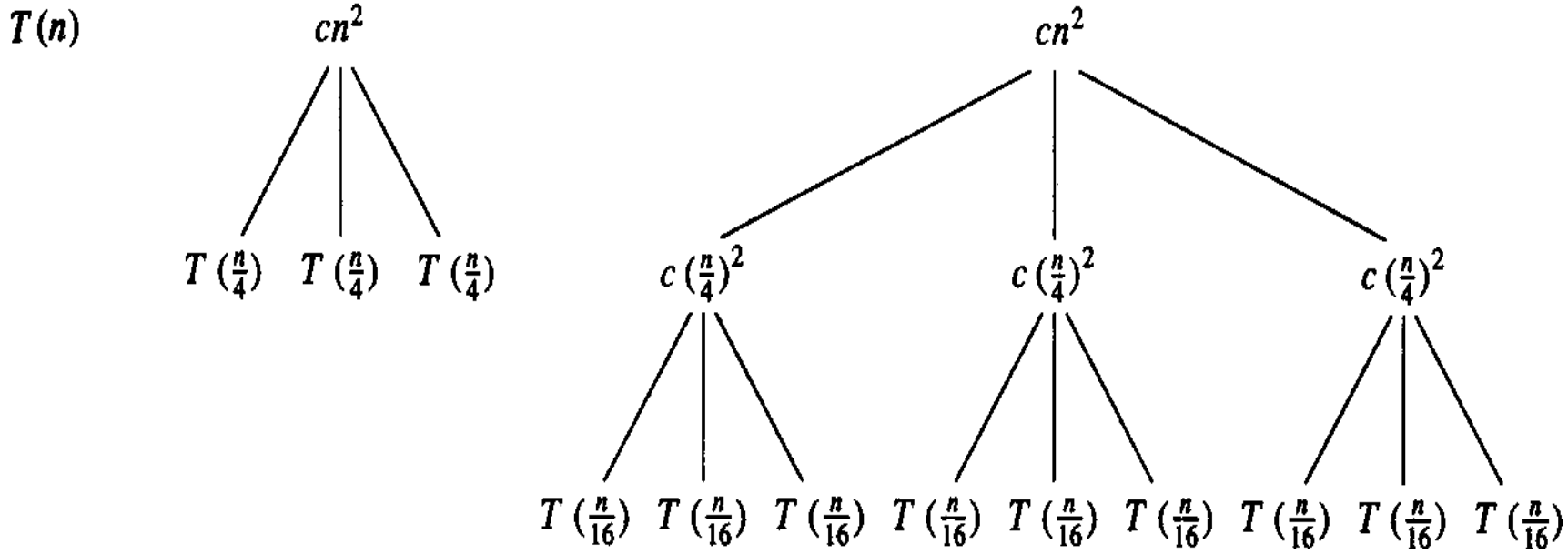
- T(n)=**$c(5/16)^0n^2 + c(5/16)^1n^2 + c(5/16)^2n^2 + \ldots + c(5/16)^in^2$**

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{5}{16}\right)^i cn^2 + \left(2^i\right) \leq \sum_{i=0}^{\infty} \left(\frac{5}{16}\right)^i cn^2 + \left(2^i\right) = \frac{1}{1-\frac{5}{16}} cn^2 + \left(2^{\log_2 n}\right) = O(n^2)$$

3. Total cost of each sub-problems are not equal but size of sub-problems are equal.

- **Example : $T(n) = 3\ T(n/4) + cn^2$**

Example : T(n) = 3 T(n/4) + cn^2



- Sub-problem size at level i is: $n/4^i$
- Sub-problem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level i = $c(n/4^i)^2$
- Number of nodes at level i = $3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes

$3^0\ cn^2 + 3^1 c(n/4)^2 + 3^2 c(n/16)^2 + \text{----}$

Or

$(3/16)^0\ cn^2 + (3/16)^1 cn^2 + (3/16)^2 cn^2 + \text{----}$

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

Worst case consider here

Total cost: $\Rightarrow T(n) = O(n^2)$

Assignment :-> T(n) = 4 T(n/2) +n, solve by recursion tree

The Master Method

- Based on the Master theorem.
- This approach for solving recurrences of the form

  $T(n) = aT(n/b) + f(n)$

  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, but we ignore floors and ceilings.
- Requires memorization of three cases.

The Master Theorem

Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and

Let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

$T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,
   and if $a \times f(n/b) \leq c\, f(n)$, for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Using the Master Method

- In each of three cases, we compare the function f(n) with the function $n^{\log_b a}$.
  - The larger of two functions determines the solution to the recurrence.
- For case 1, $f(n) < n^{\log_b a}$
- For case 2, $f(n) = n^{\log_b a}$
- For case 3, $f(n) > n^{\log_b a}$

Example of Master Method

- T(n)=9T(n/3)+n
  - For this recurrence we have a=9, b=3 and f(n)=n
  - Thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - We can apply case 1 $f(n) < n^{\log_b a} \Rightarrow n < n^2$
  - Since $f(n) = O(n^{\log_3 9-\varepsilon})$, where $\varepsilon=.1$
  - The solution is $T(n)= \Theta(n^2)$
- T(n)=T(2n/3)+1,
  - in which a=1, b=3/2 and f(n)=1
  - We have $n^{\log_b a} = n^{\log_{3/2} 1} = n^0=1$ *($f(n)= n^{\log_b a}$)*
  - Case 2 applies ,
  - Since $f(n)= \Theta(n^{\log_{3/2} 1})= \Theta(1)$
  - The solution of this recurrence is $T(n)= \Theta(\log n)$ *($T(n) = \Theta(n^{\log_b a} \log n)$)*

Example of Master Method

- T(n)=3T(n/4)+nlogn
  - We have a=3, b=4 and f(n)=nlogn
  - Thus we have that $n^{\log_b a} = n^{\log_4 3} = \mathbf{O}(n^{0.793})$
  - Since $f(n) = \mathbf{O}(n^{\log_4 3+\varepsilon})$, where $\varepsilon \approx 0.207$
  - $f(n) > n^{\log_b a} \Rightarrow n\log n > n$
  - case 3 applies if we can show that the regularity condition holds for f(n).
  - For sufficiently large n, we have that
  - af(n/b)<=cf(n)
    - af(n/b)=3f(n4)=3(n/4)log(n/4)=(3/4)nlog(n/4)
  - (¾)nlog(n/4)<=c nlogn for c=3/4 :: (c<1)
  - The solution of this recurrence is $T(n)= \mathbf{\Theta}(n\log n)$

- $T(n)= 2T(n/2) + n \log n$
- $a=2, b=2$, $f(n)= n\log n$ and $n^{\log_b a} = n$
- $n\log n > n$ case three apply (mistake)

  - For sufficiently large n, we have that
  - $af(n/b)<=cf(n)$
  - $af(n/b)=2(n/2)\log(n/2)<=(2/2)n\log n=cf(n)$ for $c=2/2=>1$ :: $(c<1)$
  - Not hold the condition

Problems

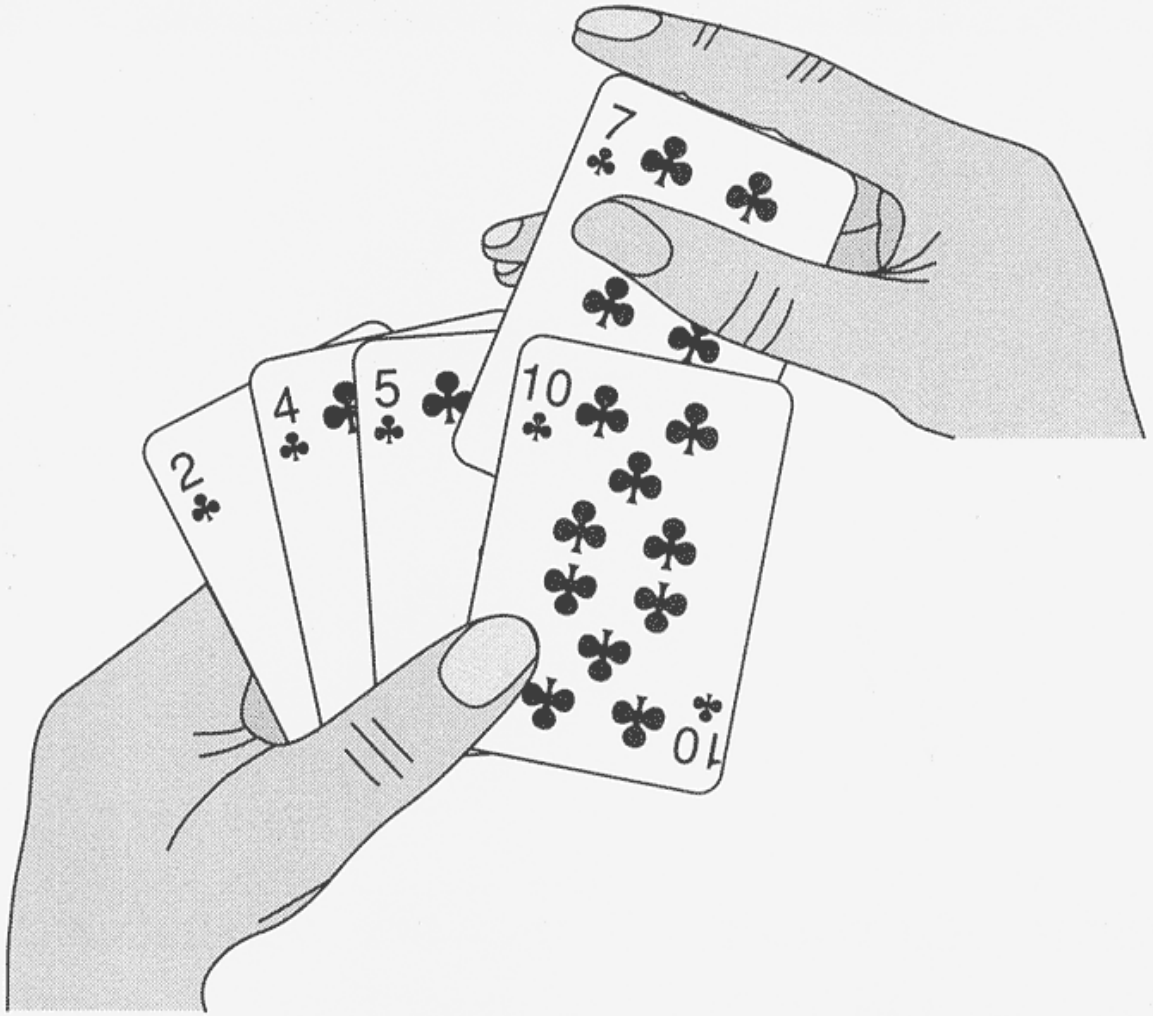
1. $T(n) = 2T(n/2) + \Theta(n)$
2. $T(n) = 8T(n/2) + \Theta(n^2)$
3. $T(n) = 7T(n/2) + \Theta(n^2)$
4. $T(n) = T(n/2) + 1$
5. $T(n) = 4T(n/3) + cn^2$
6. $T(n) = 2T(n/4) + 1$
7. $T(n) = 2T(n/4) + \sqrt{n}$
8. $T(n) = 2T(n/4) + n^2$
9. Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Solve the recurrence relation for $T(1)=O(1)$

$T(n) = 128T(n/2) + \log^3 n$

where n>=2 and a power of 2.

Insertion Sort



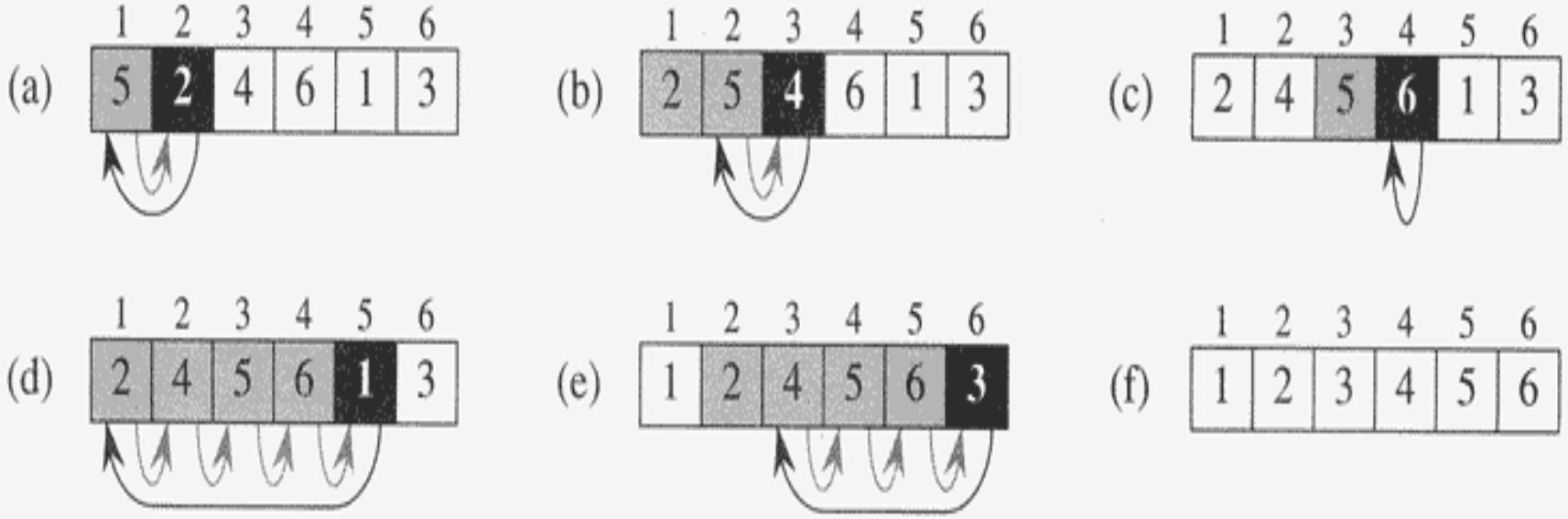


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

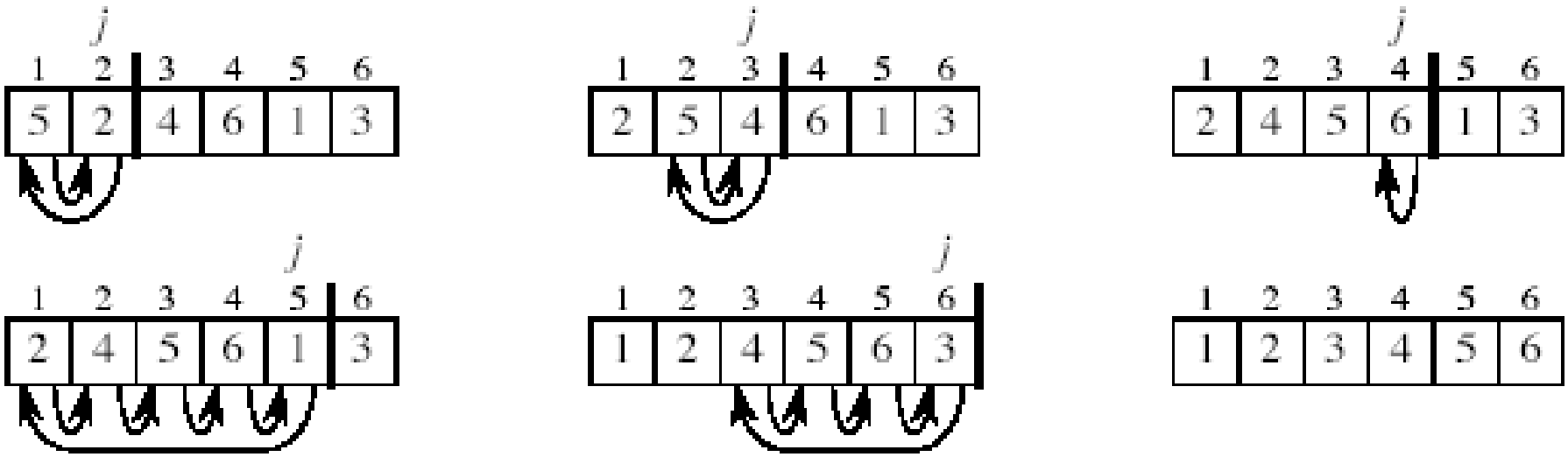
```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Procedure

- A good algorithm for sorting a small number of elements.
- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```



Problem

- Sort following list by Insertion sort.
- 4, 3, 2, 10, 12, 1, 5, 6

complexity

- Best Case T(n)= O(n)
- Average Case T(n)= $O(n^2)$
- Worst Case T(n)= $O(n^2)$

Analysis of Insertion Sort

- Assume that the ith line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the while loop test is executed for that value of j.
- Note that when a for or while loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let $T(n) =$ running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

The running time depends on the values of t_j. These vary according to the input.

Best Case

Best case: The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the while loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n.

Worst Case

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an *arithmetic series*, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Average Case

- J=J/2

complexity

- Best Case T(n)= O(n)
- Average Case T(n)= $O(n^2)$
- Worst Case T(n)= $O(n^2)$

- void InsertionSort(int a[], int n)
- {int i, j, key;
  - for(j=1; j<n; j++)

    { key =a[j];
    - i = j-1;
    - while (i>= 0) && (A[i] >key)
      - A[i+1] = A[i];
      - i = i – 1;
    - A[i+1] = key;

    }

Introduction to Algorithms Second Edition by

Cormen, Leiserson, Rivest & Stein

Chapter 7

Description of Quick Sort

- Quick sort, like merge sort, is based on divide-and-Conquer paradigm. Here is three steps for sorting a Array A[p…r]
- **Dividing:** Partition the array into two possible (possible empty) subarrays A[p..q-1] and A[q+1..r].
- **Conquer:** Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p..r] is now sorted.

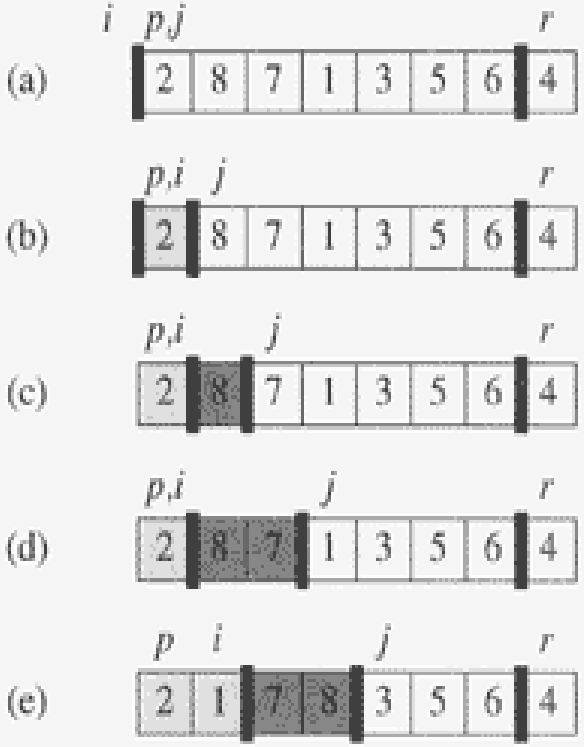
Quick sort steps

j=p, i=j-1

1. Compare pivot & jth element
2. If jth element is small than
    1. Increment in i by 1
    2. Exchange ith & jth element
3. Repeat steps 1, 2 & 4 until j reaches to last or r+1 location
4. After tracing all elements in array
5. Exchange pivot element with i+1 position element.

```
QUICKSORT(A, p, r)
1   if p < r
2       then q ← PARTITION(A, p, r)
3            QUICKSORT(A, p, q − 1)
4            QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

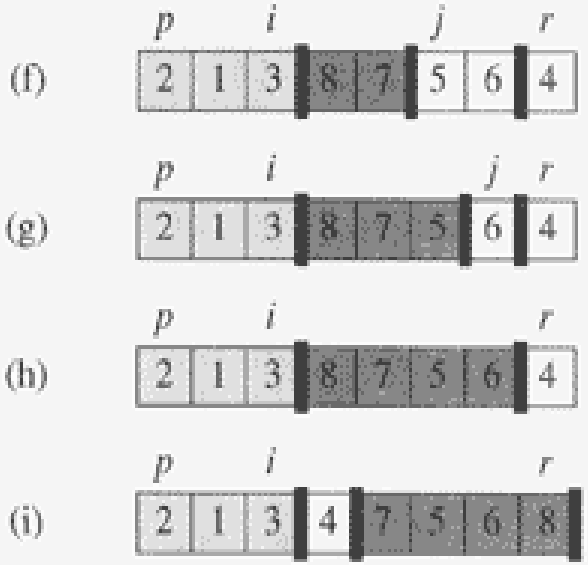


Figure 7.1 The operation of PARTITION

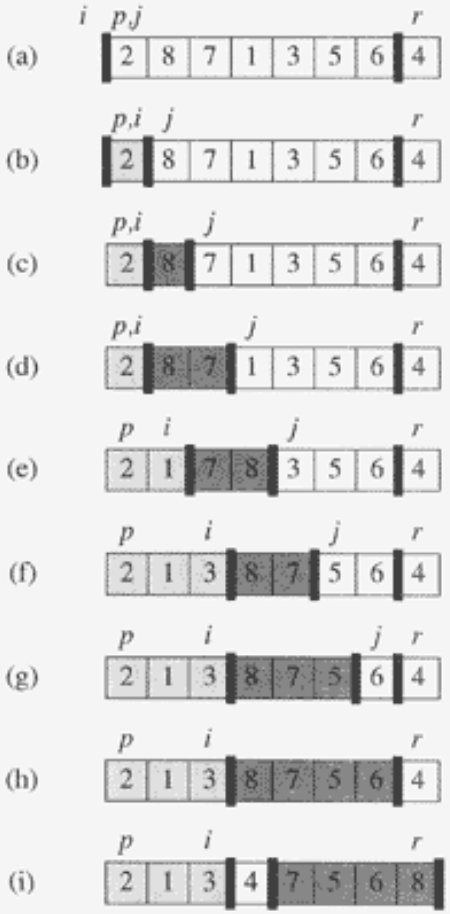


Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 8 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between

Sort the following sequence by using the Quick sort: { 10, 2, 1, 5, 3, 8, 11, 24, 7 }.

Performance of Quick Sort

The running time of quick sort is depends on whether the partitioning is balanced or unbalanced.

If the partitioning is balanced, the algorithms runs as fast as merge sort.

If the partitioning is unbalanced, however, it can run as slowly as insertion sort.

We will analyze quick sort in all three cases:

see in next slide

Worst Case Partitioning

The Worst-Case behavior for quick sort occurs when the partitioning routine products one sub-problem with n-1 elements and one with 0 element.

Let us assume that this unbalanced partitioning arise in each recursive call.

The portioning cast is $\Theta(n)$.

Since the recursive call on array of size 0 just returns, $T(0)= \Theta(1)$, the recurrence for the running time is

$T(n)= T(n-1)+T(0)+ \Theta(n)$ when $n>1$

$T(n)= T(n-1)+\Theta(n)$

$T(n)= T(n-1)+ cn$ solve by iterative method

$T(n)= T(n-1)+ cn$----------1

Put $n=n-1$ in eq^n 1st

We get $T(n-1)= T(n-2)+c(n-1)$

Put $T(n-1)$ in eq^n 1st

- $T(n)= T(n-2)+c(n-1)+cn$---------2
- put $n=(n-2)$ in eq^n 1st
- We get $T(n-2)=T(n-3)+c(n-2)$
- put in eq^n 2
- $T(n)=T(n-3)+c(n-2)+c(n-1)+cn$
- $T(n)= T(n-3)+3cn-3c$
- ----------
- $T(n)= T(n-k)+kcn-kc$ :: $k=n-1$
- $T(n)=T(n-(n-1))+[(n-1)cn]+c(n-1)$
- $T(n)=T(1)+cn^2-cn+cn-c$
- $T(n)=\Theta(1)+cn^2-c$
- **$T(n)=\Theta(n^2)$**

Best Case Partitioning

- In the most even possible spilt, PARTITION produces two sub-problems, each of size no more than n/2:
  - $T(n)=2T(n/2)+\Theta(n)$
  - $T(n)=\Theta(n\log n)$

Average Case Partitioning

- Partitioning ratio is 9:1
- $T(n)=T(9n/10)+T(n/10)+ \Theta(n)$

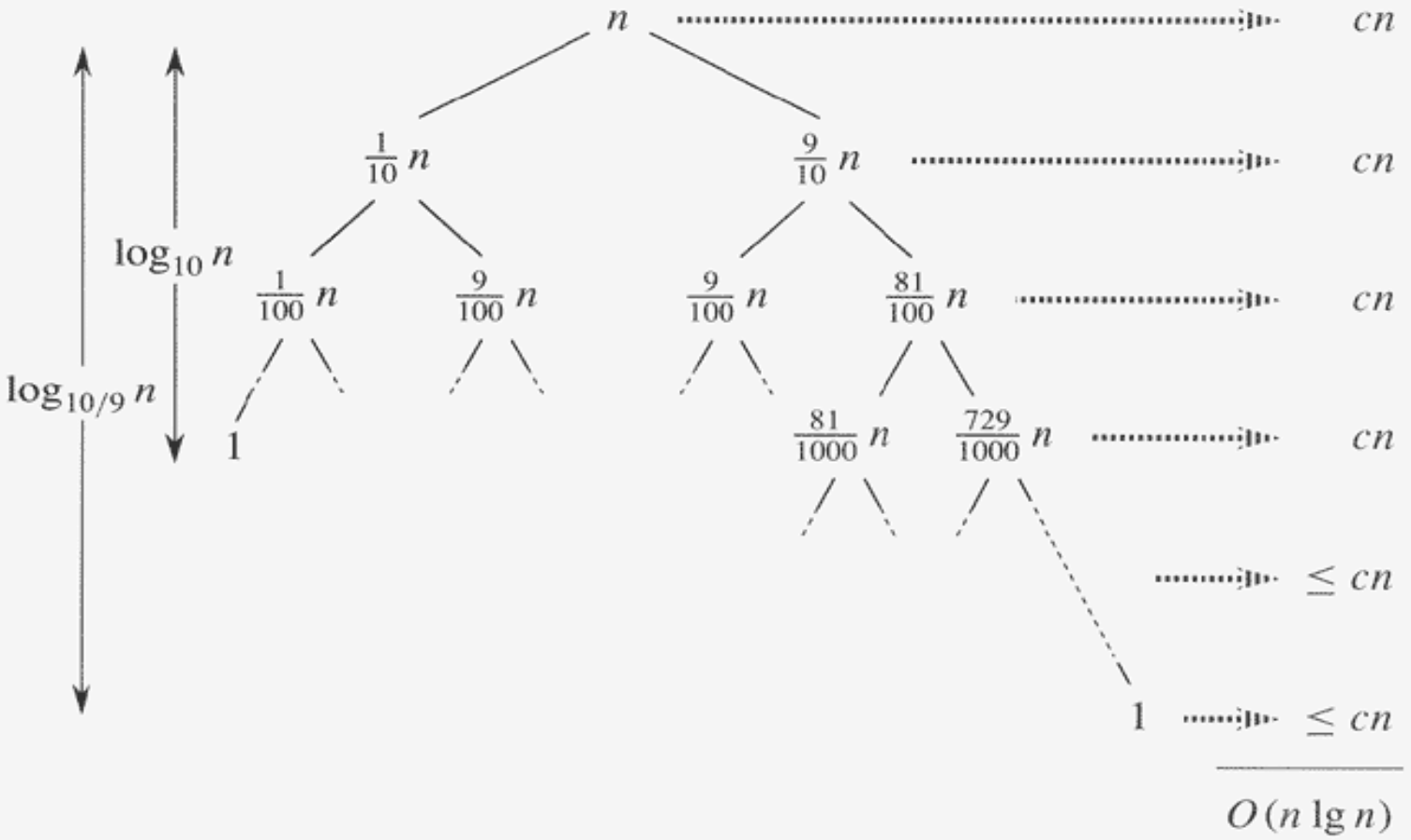


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Size of subproblem at i^{th} level $(9/10)^i n$

$i=\log_{10/9} n$

Cost of each level is cn

T(n) <= cost of each level × Height of longest Tree

<= $cn \times (\log_{10/9} n+1)$

$T(n)=O(n\log_{10/9} n)$

Or

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

A Randomized version of quick

```
RANDOMIZED-PARTITION(A, p, r)
1.    i = Random(p, r)
2.    Exchange A[r] with A[i]
3.    Return PARTITION(A, p, r)
```

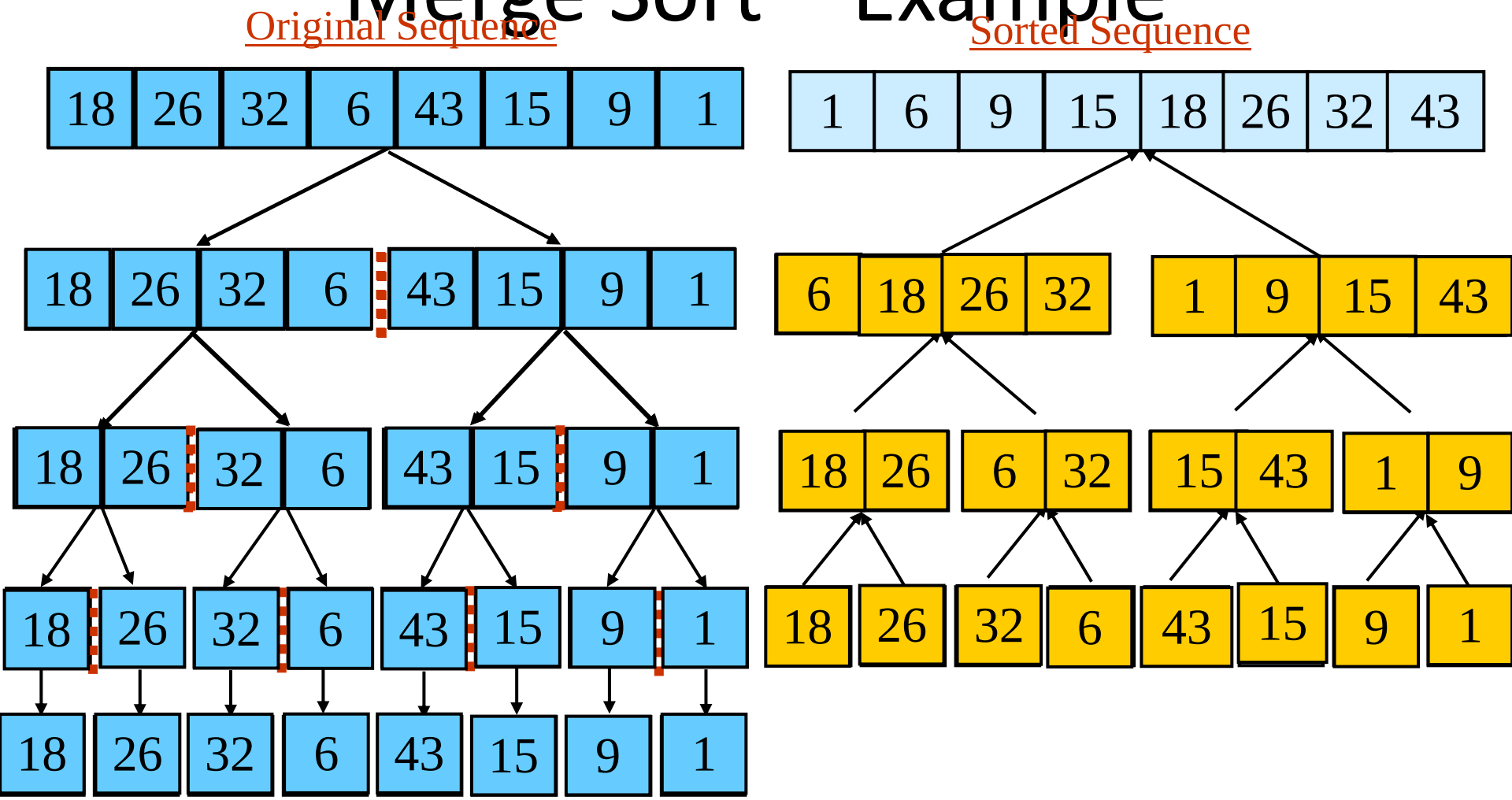
```
RANDOMMIZED-QUICKSORT(A, p, r)
1.    if p < r
2.          q = RANDOMIZED-PARTITION(A, p, r)
3.          RANDOMMIZED-QUICKSORT(A, p, q-1)
4.        RANDOMMIZED-QUICKSORT(A, q+1, r)
```

An Example: Merge Sort

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine*:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1   if p < r
2       then q ← ⌊(p+r)/2⌋
3            MergeSort (A, p, q)
4            MergeSort (A, q+1, r)
5            Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Procedure Merge

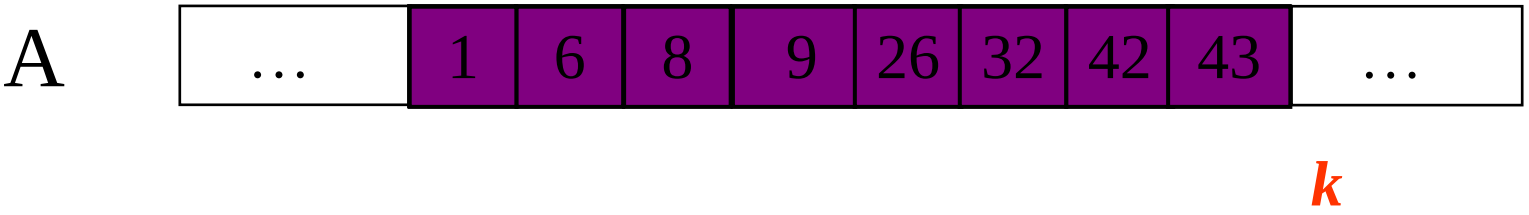
```
Merge(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3      for i ← 1 to n1
4          do L[i] ← A[p + i − 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13            then A[k] ← L[i]
14                 i ← i + 1
15            else A[k] ← R[j]
16                 j ← j + 1
```

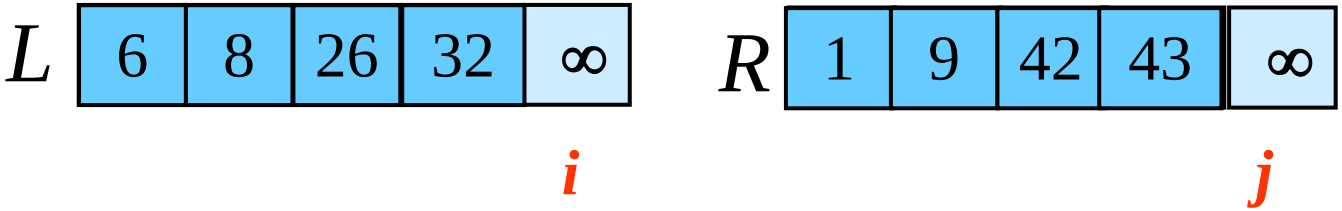
Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge – Example





Correctness of Merge

```
Merge(A, p, q, r)
1  n1 ← q – p + 1
2  n2 ← r – q
3      for i ← 1 to n1
4          do L[i] ← A[p + i – 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ←p to r
12         do if L[i] ≤ R[j]
13             then A[k] ← L[i]
14                  i ← i + 1
15             else A[k] ← R[j]
16                  j ← j + 1
```

Loop Invariant for the *for* loop

At the start of each iteration of the for loop:

Subarray $A[p..k-1]$ contains the $k-p$ smallest elements of L and R in sorted order.

$L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization:

Before the first iteration:

- $A[p..k-1]$ is empty.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Correctness of Merge

```
Merge(A, p, q, r)
1  n1 ← q – p + 1
2  n2 ← r – q
3      for i ← 1 to n1
4          do L[i] ← A[p + i – 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13            then A[k] ← L[i]
14                 i ← i + 1
15            else A[k] ← R[j]
16                 j ← j + 1
```

Maintenance:

Case 1: $L[i] \le R[j]$

- By LI, A contains $p - k$ smallest elements of L and R in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $p - k + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for $L[i] > R[j]$.

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest elements of L and R in sorted order.
- L and R together contain $r - p + 3$ elements. All but the two sentinels have been copied back into A.

Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

$$\Rightarrow T(n) = \Theta(n \lg n)$$

Example – Exact Function

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:** $T(n) = 2\,T(n/2) + n$

$$= 2\,((n/2)\lg(n/2) + (n/2)) + n$$

$$= n\,(\lg(n/2)) + 2n$$

$$= n \lg n - n + 2n$$

$$= n \lg n + n$$

Recursion-tree Method

- Making a good guess is sometimes difficult with the substitution method.
- Use **recursion trees** to devise good guesses.
- Recursion Trees
  - Show successive expansions of recurrences using trees.
  - Keep track of the time spent on the subproblems of a divide and conquer algorithm.
  - Help organize the algebraic bookkeeping necessary to solve a recurrence.

Recursion Tree – Example

- Running time of Merge Sort:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

- Rewrite the recurrence as

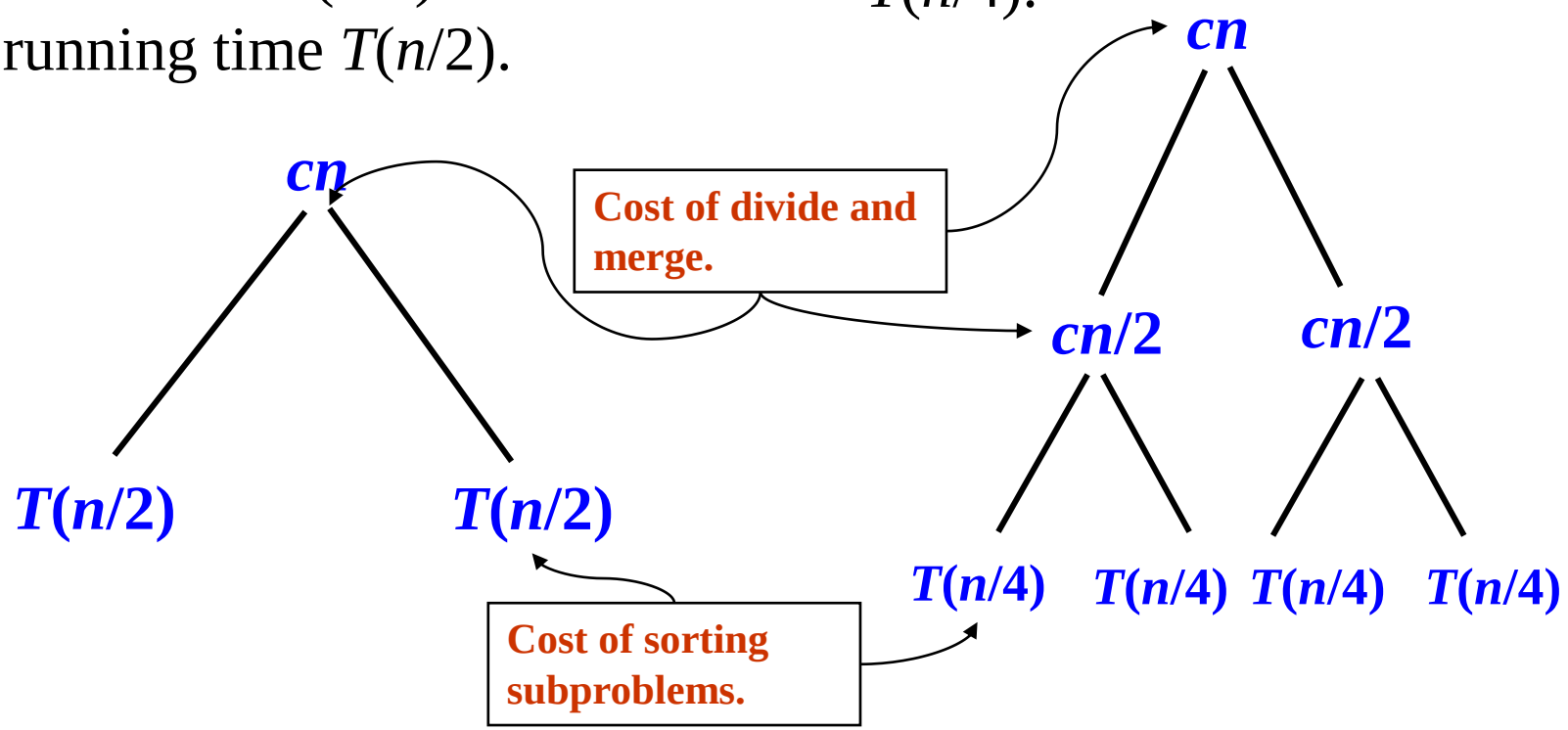
$$T(n) = \mathbf{c} \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \mathbf{cn} \quad \text{if } n > 1$$

$c > 0$: Running time for the base case and time per array element for the divide and combine steps.

Recursion Tree for Merge Sort

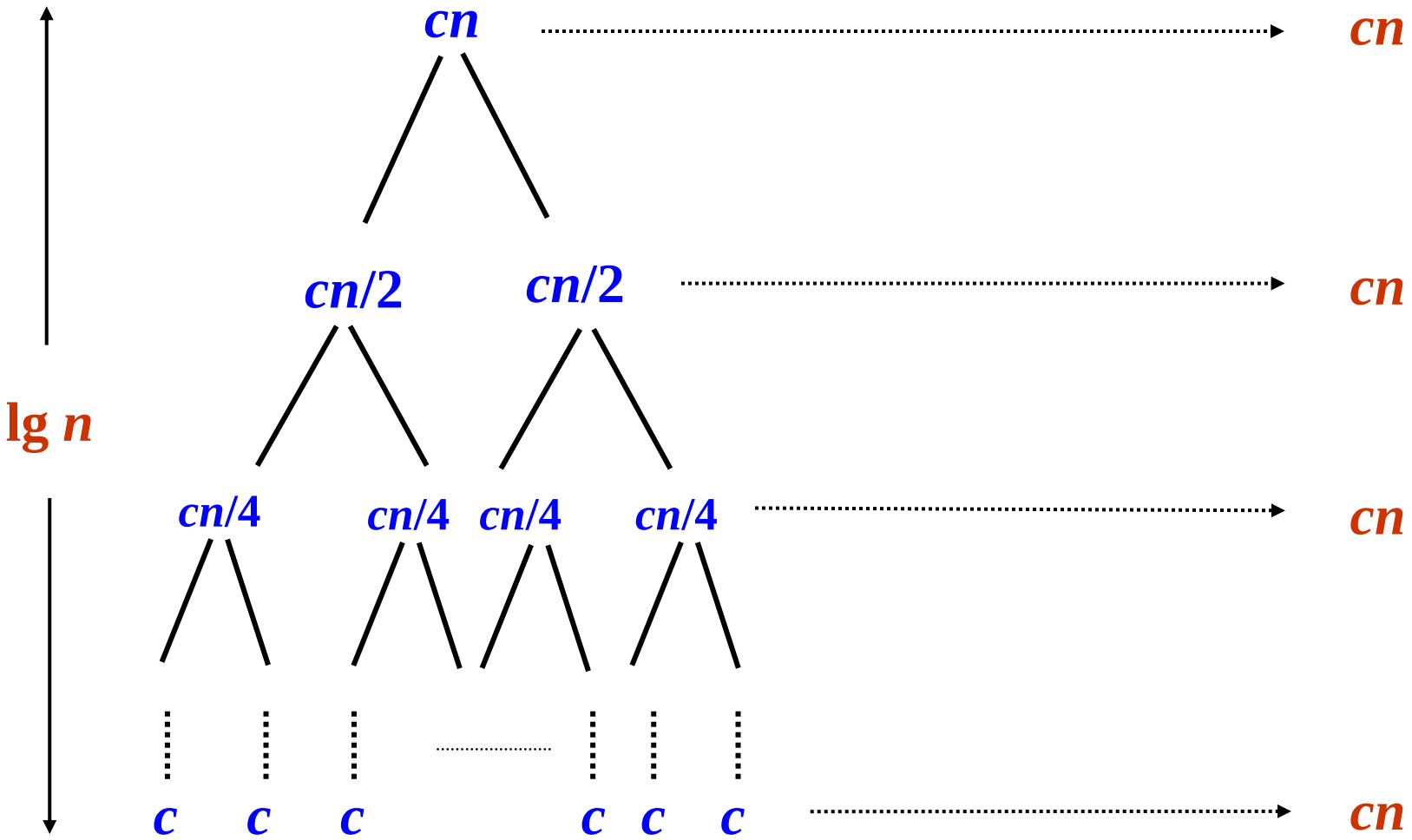
For the original problem, we have a cost of *cn*, plus two subproblems each of size (*n*/2) and running time *T*(*n*/2).

Each of the size *n*/2 problems has a cost of *cn*/2 plus two subproblems, each costing *T*(*n*/4).



Recursion Tree for Merge Sort

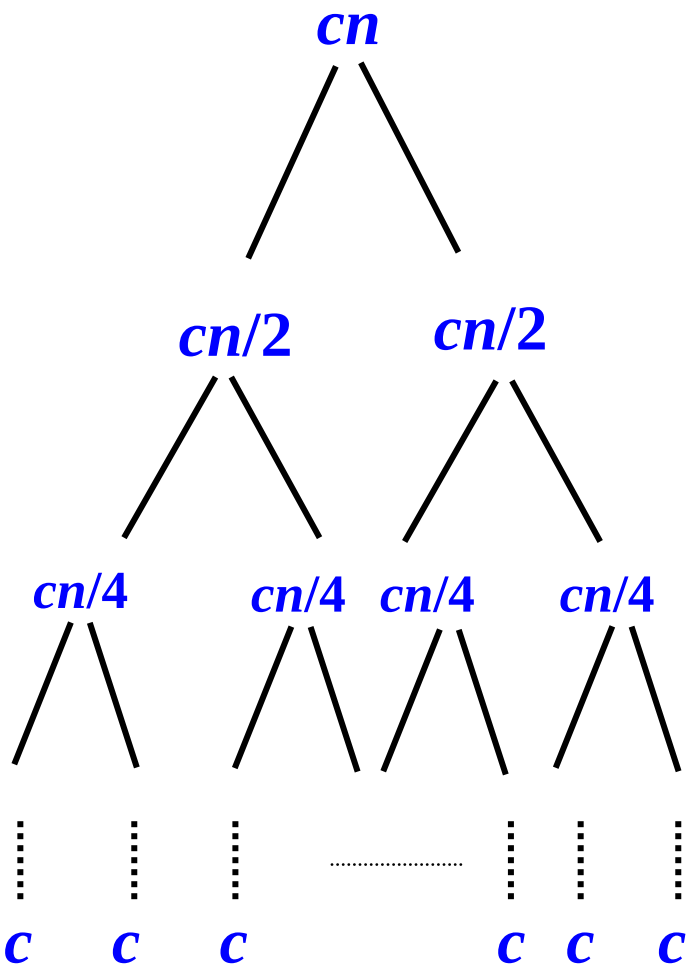
Continue expanding until the problem size reduces to 1.



Total : $cn\lg n + cn$

Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves ⇒ *cost per level remains the same*.
- There are lg n + 1 levels, height is lg n. (Assuming n is a power of 2.)
  - Can be proved by induction.
- Total cost = sum of costs at each level = $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$.

Recursion Trees – Caution Note

- Recursion trees only generate guesses.
  - Verify guesses using substitution method.
- A small amount of "sloppiness" can be tolerated. Why?
- If careful when drawing out a recursion tree and summing the costs, can be used as direct proof.


Thank You