Analysis of Algorithms

COMP.4040, Summer 2019

Chapter 2: Analyzing and Designing Algorithms

By: Sirong Lin, PhD

University of Massachusetts Lowell

UMASS LOWELL

Learning with Purpose

Outline

Introduce the framework of analyzing and designing algorithms

Examine two sorting algorithms: insertion sort and merge sort

- pseudocode
- asymptotic notations
- divide and conquer approach

Framework of Analyzing Algorithms

Framework

define the problem

develop the algorithm (write pseudocode)

  correctness

analyze the running time using notations

revise the algorithm

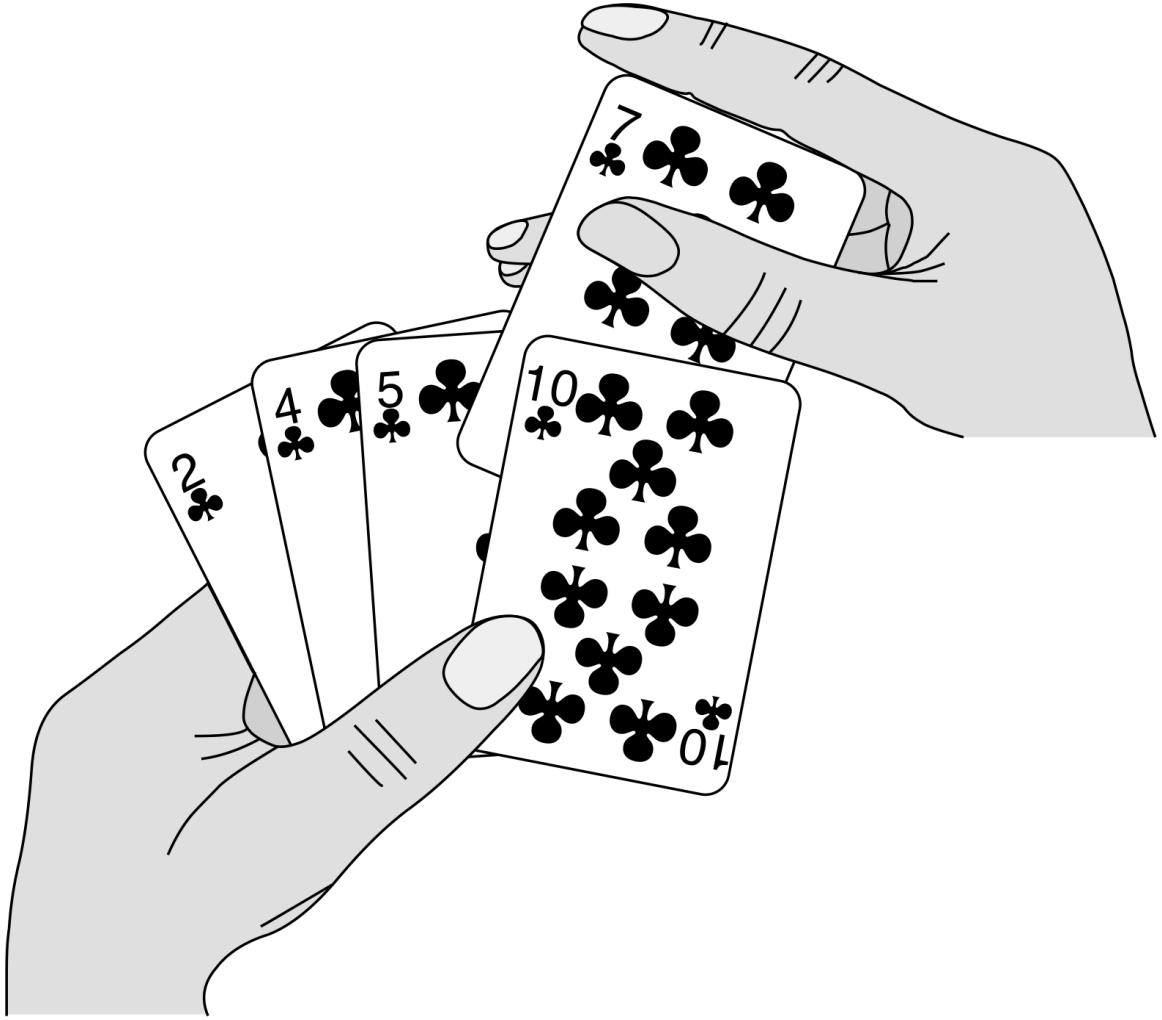
Problem of Sorting

Input, output

many algorithms to solve the problem, but the performance is different

- slow algorithms
- fast algorithms

Insertion Sort



Insertion Sort Pseudocode

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      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          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Pseudocode Conventions

Indentation indicates block structure

use **while**, **for**, **repeat-until** for loops

// indicates comment

variables, such as *i*, *j*, and *key* are local variables

convention: capital letters A — array, matrix, n — the size of array, etc.

logic “and” and “or” are ***short circuiting***

Pseudocode Conventions (Cont'd)

access **array** element

- specify the array name followed by index in square brackets, e.g., A[i]
- index starts from 1, but not 0
- .. indicates the a range within an array, e.g., A[1..j]

an attribute of an object, e.g., *A.length* (the length of the array)

Pseudocode Conventions (Cont'd)

use **if-else** for conditional structure

pass parameters to a procedure ***by value***

- arrays are passed by pointer (instead of the entire array)

multiple assignment: i = j = e (i.e., j = e, followed by the assignment i = j)

Pseudocode Conventions (Cont'd)

return statement immediately transfer control back to the calling procedure

keyword **error** indicates an error occurred

Pseudocode Conventions (Cont'd)

General Rule of Writing Pseudocode —

make an algorithm as short as possible

Please use textbook pseudocode convention in homework, quizzes and tests

Insertion Sort Pseudocode

How does the algorithm works?

example in notes

How do we know if the algorithm is correct?

Loop Invariants

Loop invariant — property of a program loop that will hold for the following 3 conditions: initialization, maintenance, termination of iterations

help us to understand why an algorithm is correct

Loop Invariants (Cont'd)

Initialization: It is true prior to the first iteration of the loop

Maintenance: it remains true before each next iteration

Termination: when the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Loop Invariants (Cont'd)

Like mathematical induction — to prove that a property holds, prove a base case and an inductive step

- Showing that the invariant holds *before* the first iteration — base case
- Showing that the invariant holds *from iteration to iteration* — the inductive step
- We stop the "induction" when the loop terminates (which is different from "inductive step is used infinitely in math")

Loop Invariant for Insertion Sort

What is the loop invariant for insertion sort?

At the start of each iteration of the for loop (line 1~8), the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order

Loop Invariant for Insertion Sort (Cont'd)

Initialization: j == 2, subarray consists A[1]

Maintenance: the while loop makes the loop invariant hold

Termination: when the loop terminates, j == n+1, the entire array consists all elements, in sorted order

Our insertion sort algorithm is correct!

How good is Insertion Sort

depends on input itself

- array initially sorted in ascending order
- array initially sorted in descending order

also depends on the input size (the number of elements)

- 6 vs. 6×10^9

Analysis of Algorithms

Analysis of Algorithm

Algorithm running time: predict the resources that the algorithm requires

- most often we measure computational time (what we will mainly focus on)
- other measurement: memory, band-width, hardware

Analysis of Algorithm (Cont'd)

How do we analyze an algorithm's running time?

- depends on the input
  - larger input size does takes longer time
  - two inputs of the same size may take different amounts of time

Analysis of Algorithm (Cont'd)

Input size:

- the number of items in the input, e.g., the array size *n* for sorting
- the total number of digits, e.g., multiplying two integers
- two numbers, e.g., # of vertices and edges in a graph

Analysis of Algorithm (Cont'd)

Running Time: the number of primitive "operations" or "steps" executed

steps: machine-independent

assumptions:

$\mathbf{c_i}$: the constant time that executing the i^{th} line

each line consists only of primitive operations

Analysis of Insertion sort

T(n): running time of Insertion_Sort

n: the size of the array (input size)

c_i: the i^{th} line takes time c_i

t_j: number of times that while loop test is executed for that value of j

Analysis of Insertion sort

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

Analysis of Insertion sort (Cont'd)

Running Time: (in general)

t_j: number of times that while loop test is executed for that value of j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \; .$$

Analysis of Insertion sort (Cont'd)

Running Time:

best case (sorted, in ascending order): $t_j = 1$

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

linear function of n

Analysis of Insertion sort (Cont'd)

Running Time:

worst case: reversed sorted $t_j = j$

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

quadratic function of n

Analysis of Insertion sort (Cont'd)

Which time to use?

- usually worst-case running time
  - guarantee the upper bound on the running time for any input (e.g., no more than 3 seconds)
- why not average-case?

$$t_j \approx j/2$$

  - still a *quadratic function* of n
  - it is often as bad as the worst case

Analysis of Algorithms (Cont'd)

worst-case (usually)

- T(n) is the max time on any input of size n

average-case (sometimes)

- T(n) is expected time over all input of size n
- it is the probability of any given situation
- hard to predict, need to make assumption of statistical distribution of input

best-case (not often)

- can cheat to say an algorithm (slow one) works well for a particular input

Homework 1

Due Date: May 23 (Th), BEFORE the lecture starts

- Start working on it as we learn
- Honor Statement needs to be enclosed for each assignment, otherwise the homework will not be graded

Order of Growth
— Asymptotical Analysis

Order of growth

Some simplifications we made:

- c_i: cost to execute each line of code is constant
- T(n): $an^2 + bn + c$ (a, b, c are some constants)

Need more abstraction

- how can we compare the *linear* function of n with the *quadratic* function of n? and many other functions of n?

Order of growth (Cont'd)

BIG IDEA: **Asymptotical Analysis**

rate of growth/order of growth as n —> ∞

Only look at the *leading term* of the formula for running time

Drop lower-order terms

Ignore the constant coefficient in the leading term

Order of growth for Insertion sort

T(n): $an^2 + bn + c$

Drop lower-order terms => an^2

Ignore the constant coefficient in the leading term => n^2

The running time of Insertion_Sort *grows like* n^2 (can't say, it equals to n^2)

the order of growth is n^2.

Order of growth (Cont'd)

One algorithm to be more efficient than another if its worst-case running time has a smaller order of growth

e.g., a $\Theta(n^2)$ algorithm will run more quickly in the worst case than a $\Theta(n^3)$ algorithm

But is this always true?

Order of growth (Cont'd)

A $\Theta(n^2)$ algorithm will run more quickly in the worst case than a $\Theta(n^3)$ algorithm (always true?)

- true for large enough inputs (as n —> ∞)
- not necessarily mean that a $\Theta(n^3)$ algorithm is always slower than a $\Theta(n^2)$ algorithm

Order of growth (Cont'd)

We shouldn't consider the slower algorithms at all. (True or False?)

- still interested in the slower algorithms
  - They may be asymptotically slower, but they are fast for reasonably sized data
- need to find a balance
  - mathematically understanding
  - engineering common sense

Order of growth for Insertion sort (Cont'd)

Is Insertion sort fast?

moderate fast for small n

not at all for large n

Can we do better than Insertion Sort (worst-case) to solve the problem of Sorting?

(We will ask this kind of question all the time :))

Designing Algorithms

Approaches of Designing Algorithms

Incremental approach:

A[1.. j-1] sorted, place A[j] correctly, so that A[1..j] is sorted

Approaches of Designing Algorithms (Cont'd)

Divide and Conquer approach:

- Divide: the problem into small subproblems
- Conquer: the subproblems by solving them **recursively**
- Combine: the sub-solutions to solve the original problem

Merge Sort (an example of D&C)

The idea of Merge Sort (notes)

Merge Sort — Pseudocode

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

Key Operation

Initial call: MERGE_SORT(A, 1, n)

Merge (key operation)

Input: Array A, indices p, q, such that

p<= q < r

subarrays A[p..q] and A[q+1..r] both sorted

Output: two subarrays are *merged into a single sorted subarray A[p..r]*

Goal: to achieve Θ(n) of Running Time, n = r-p+1 (n is the size of the subproblem)

Merge (key operation) (Cont'd)

Step 1:

copy A[p..q] into a temp array L[1..n_1+1], where n_1 = q-p+1 and L[n_1+1] stores a sentinel value (∞)

copy A[q+1..r] into a temp array R[1..n_2+1], where n_2 = r-q and R[n_2+1] stores a sentinel value (∞)

Step 2:

merge two subarrays L and R back into A[p..q]

A[p..q] is sorted

Merge - Pseudocode

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Merging (key operation)

Loop invariant for line 12-17

subarray A[p..k-1] contains k-p smallest elements of L & R, in sorted order

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

check the *initialization*, *maintenance*, and *termination* properties for the invariant by yourself (p32~33)

Merging (key operation)

Running Time of Merging:

first two *for* loops (line 4-5, 6-7)

$\Theta(n_1 + n_2) = \Theta(n)$

last *for* loop (line 12-17)

n iterations, each line constant time, so $\Theta(n)$

Total time: $\Theta(n)$

Analyzing Merge_Sort (D&C algorithms)

Running time T(n) contains 3 parts:

Divide: D(n)

compute q, => Θ(1) (constant time)

Conquer: recursively solve 2 subproblems, each subproblem is size n/2,

T(*n/2*): time to solve one subproblem, size of n/2

=>2*T(n/2)* **Recurrence**

Combine: **C(n)**

Merge => Θ(n)

Analyzing Merge_Sort (D&C) (Cont'd)

use ***recurrence*** to describe the running time of a divide-and-conquer algorithm

recurrence: an equation or inequality that describes a function in terms of its value on smaller inputs, e.g.,

T(n/2), T(n/4), T(n/3)

Analyzing Merge_Sort (D&C) (Cont'd)

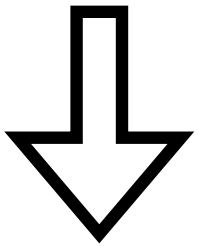
Merge Sort:

assume n is a power of 2

subarray size is exactly *n/2* all the time for merge_sort

Analyzing Merge_Sort (D&C) (Cont'd)

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



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

Solving the Merge_Sort recurrence

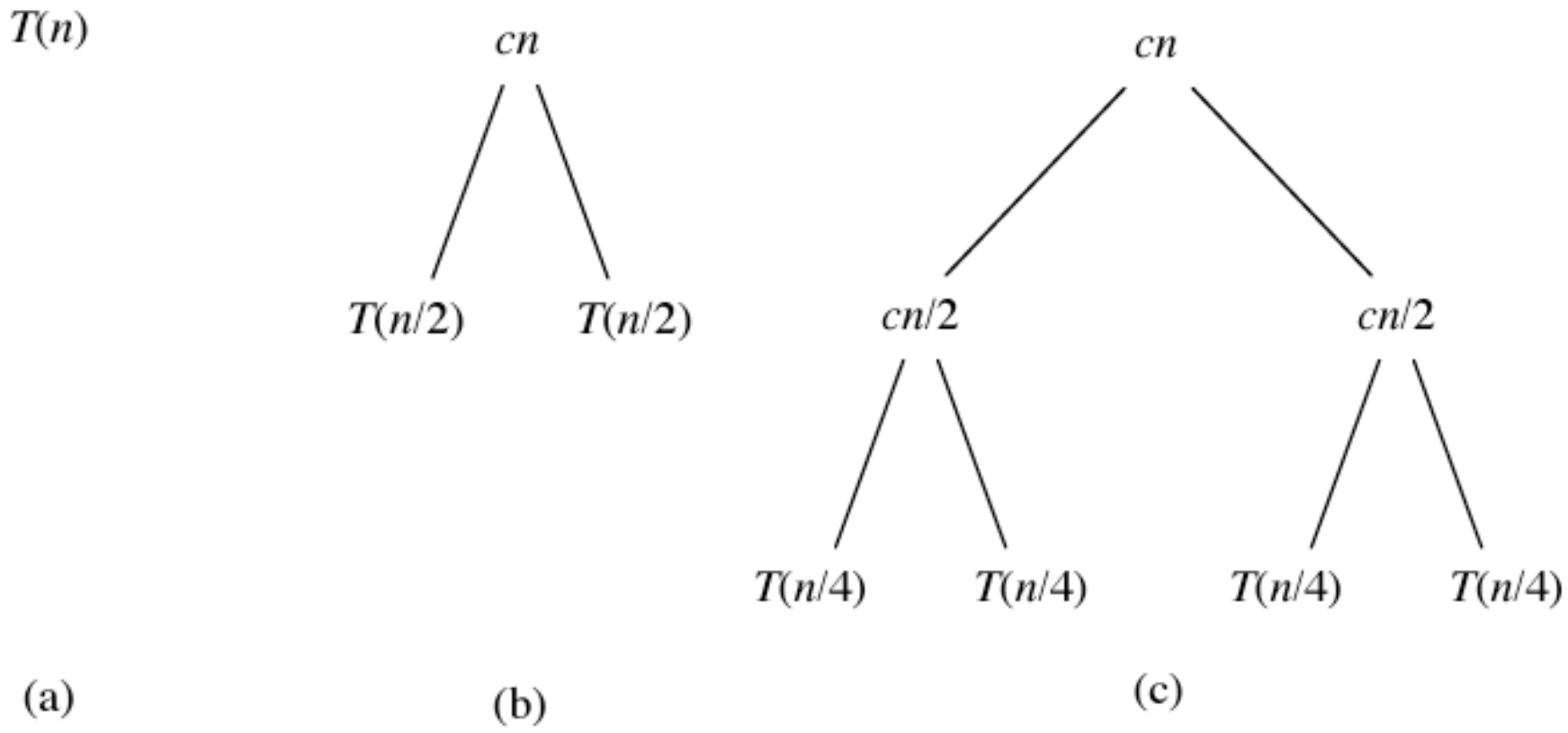
How can we solve $T(n) = 2T(n/2) + cn$ $(n>1)$?

Method 1: Recursion Tree (a visual way of solving)

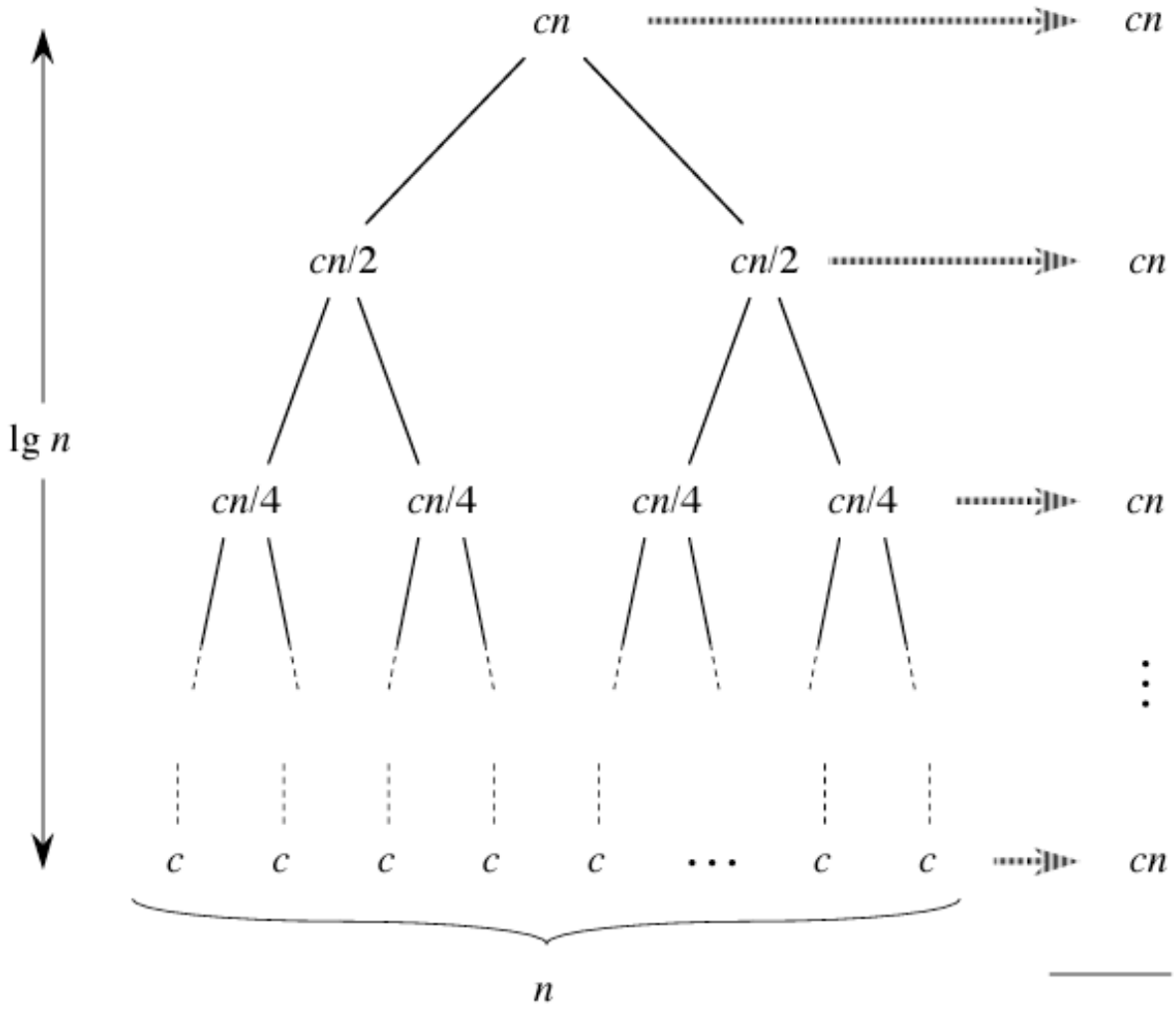
Method 2: Master Theorem (Chapter 4)

Solving the Merge_Sort recurrence — Recursion Tree

T(n) = 2T(n/2) + cn



Solving the Merge_Sort recurrence — Recursion Tree (Cont'd)



(d)

Solving the Merge_Sort recurrence — Recursion Tree (Cont'd)

Cost for each level: cn

# of Levels: $\lg n + 1$ (height: $\lg n$), where $\lg n = \log_2 n$

can be proved by induction, p37 in textbook

$T(n) = cn\,(\lg n+1) = cn \lg n + cn \Rightarrow \Theta(n \lg n)$

Solving the Merge_Sort recurrence — Master Theorem

by using Master Theorem in Chapter 4, we can also get $T(n) = \Theta(n \lg n)$

Compare Merge_Sort vs. Insertion Sort

Merge sort running time: T(n) = $\Theta(n \lg n)$

Compared to insertion sort worst-case time, $\Theta(n^2)$, merge sort is faster, asymptotically

Merge Sort **asymptotically beats** Insertion Sort

- trading a factor of n for a factor of $\lg n$ is a good deal!

In reality, Insertion sort works fine when n<30

Analyzing divide-and-conquer algorithms in general

Running time

Divide: D(n) => Θ(1) (constant time)

Conquer: recursively solve *a* subproblems, each subproblem is size *n/b*

T(*n/b*): time to solve one subproblem, size of n/b

=>aT(n/b)

Combine: **C(n)**

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

Calculate Running Time for Algorithms

T(n) = sum of (*the cost for each line* x *the number of each line executed*)

the cost for each line

constant, i.e., c_i (*i* is *i*th line)

T(*n/b*), for a recursive function call

the number of each line executed

loop: Summation $\sum$

1: if only executed once

Calculate Running Time for Algorithms

Some examples (notes)

Review

Appendix A: Summations

e.g., A.5 & A.6 for homework