Getting Started

Heejin Park
Hanyang University

Contents

- Sorting problem

- 2 sorting algorithms
  - Insertion sort
  - Merge sort

Sorting problem

keys

- **Input**
  - A sequence of n number $<a_1, a_2, . . ., a_n>$.
- **Output**
  - A permutation (reordering) $<a'_1, a'_2, . . ., a'_n>$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.
- Ex>
  - Input: < 5, 2, 4, 6, 1, 3>
  - Output: < 1, 2, 3, 4, 5, 6>

Insertion sort

- **Insertion sort**
  - Description
  - Correctness
  - Performance

Description

- What is insertion sort?
  - A sorting algorithm using insertion.

- What is insertion?
  - Given a key and a sorted list of keys, insert the key into the sorted list preserving the sorted order.
  - ex> Insert 3 into <1, 2, 4, 5, 6>

Description

- Insertion sort uses insertion incrementally.
  - Let $A[1..n]$ denote the array storing keys.
  - Insert $A[2]$ into $A[1]$.
  - Insert $A[3]$ into $A[1..2]$.
  - Insert $A[4]$ into $A[1..3]$.

    .

    .

    .
  - Insert $A[n]$ into $A[1..n\text{-}1]$.

Description: example

- 5 2 4 6 1 3
- 5 2 4 6 1 3
- 2 5 4 6 1 3
- 2 4 5 6 1 3
- 2 4 5 6 1 3
- 1 2 4 5 6 3
- 1 2 3 4 5 6

Description: pseudo code

```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
        // Insert A[j] into the sorted
              sequence A[1..j - 1].
        i = j - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Pseudocode conventions are given in p. 19 - 20 of the textbook.

n-1 iterations of insertion.

Insert $A[j]$ into $A[1..j - 1]$.

Find a place to put $A[j]$.

Put $A[j]$.

Insertion sort

- Insertion sort
  - Description
  - Correctness
  - Performance
    - Running time
    - Space consumption

Running time

- How to analyze the running time of an algorithm?
  - Consider running the algorithm on a specific machine and measure the running time.
    - We cannot compare the running time of an algorithm on a machine with the running time of another algorithm on another machine.
    - So, we have to measure the running time of every algorithm on a specific machine, which is impossible.
  - Hence, we count the number of instructions used by the algorithm.

Instructions

- Arithmetic
  - Add, Subtract, Multiply, Divide, remainder, floor, ceiling
- Data movement
  - Load, store, copy
- Control
  - Conditional branch
  - Unconditional branch
  - Subroutine call and return

Running time

- The running time of an algorithm grows with the input size, which is the number of items in the input.
- For example, sorting 10 keys is faster than sorting 100 keys.
- So the running time of an algorithm is described as a function of input size n, for example, $T(n)$.

Running time of insertion sort

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

- ***T*(*n*)**: The sum of product of *cost* and *times* of each line.

Running time of insertion sort

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

cost	*times*
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	$n - 1$

- ***T*(*n*)**: The sum of product of *cost* and *times* of each line.

Running time of insertion sort

- t_j: The number of times the **while** loop test is executed for j.

- Note that **for**, **while** loop test is executed one time more than the loop body.

Running time of insertion sort

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

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

- Although the size of the input is the same, we have
  - best case
  - average case, and
  - worst case.

Running time of insertion sort

- Best case
  - If $A[1..n]$ is already sorted, $t_j = 1$ for $j = 2, 3, \ldots, n$.

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

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

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

- This running time can be expressed as $an+b$ for *constants a* and *b*; it is thus a ***linear function*** of *n*.

Running time of insertion sort

- Worst case
  - If $A[1..n]$ is sorted in reverse order, $t_j = j$ for $j = 2, 3, \ldots, n$.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \text{and} \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

- This running time can be expressed as $an^2 + bn + c$ for constants a, b, and c; it is thus a ***quadratic function*** of n.

Running time of insertion sort

- **Only the degree of leading term is important.**
  - Because we are only interested in the ***rate of growth*** or ***order of growth.***
  - For example, a quadratic function grows faster than any linear function.

- The degree of leading term is expressed as Θ–notation.
  - The worst-case running time of insertion sort is $\Theta(n^2)$.

Space consumption of insertion sort

- $\Theta(n)$ space.

- Moreover, the input numbers are ***sorted in place*.**
  - $n + c$ space for some constant c.

Self-study on Insertion Sort

- **Exercise 2.1-1**
- **Exercise 2.1-2**

Content

- Sorting problem

- Sorting algorithms
  - Insertion sort - $\Theta(n^2)$.
  - Merge sort - $\Theta(n\lg n)$.

Merge

- What is merge sort?
  - A sorting algorithm using merge.

- What is merge?
  - Given two sorted lists of keys, generate a sorted list of the keys in the given sorted lists.
  - <1, 5, 6, 8> < 2, 4, 7, 9> → < 1, 2, 4, 5, 6, 7, 8, 9>

Merge

- Merging example
  - <1, 5, 6, 8> <2, 4, 7, 9> → < 1 >
  - < 5, 6, 8> <2, 4, 7, 9> → < 1, 2 >
  - < 5, 6, 8> < 4, 7, 9> → < 1, 2, 4 >
  - < 5, 6, 8> < 7, 9> → < 1, 2, 4, 5 >
  - < 6, 8> < 7, 9> → < 1, 2, 4, 5, 6 >
  - < 8> < 7, 9> → < 1, 2, 4, 5, 6, 7 >
  - < 8> < 9> → < 1, 2, 4, 5, 6, 7, 8 >
  - < > < 9> → < 1, 2, 4, 5, 6, 7, 8, 9>

Merge

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

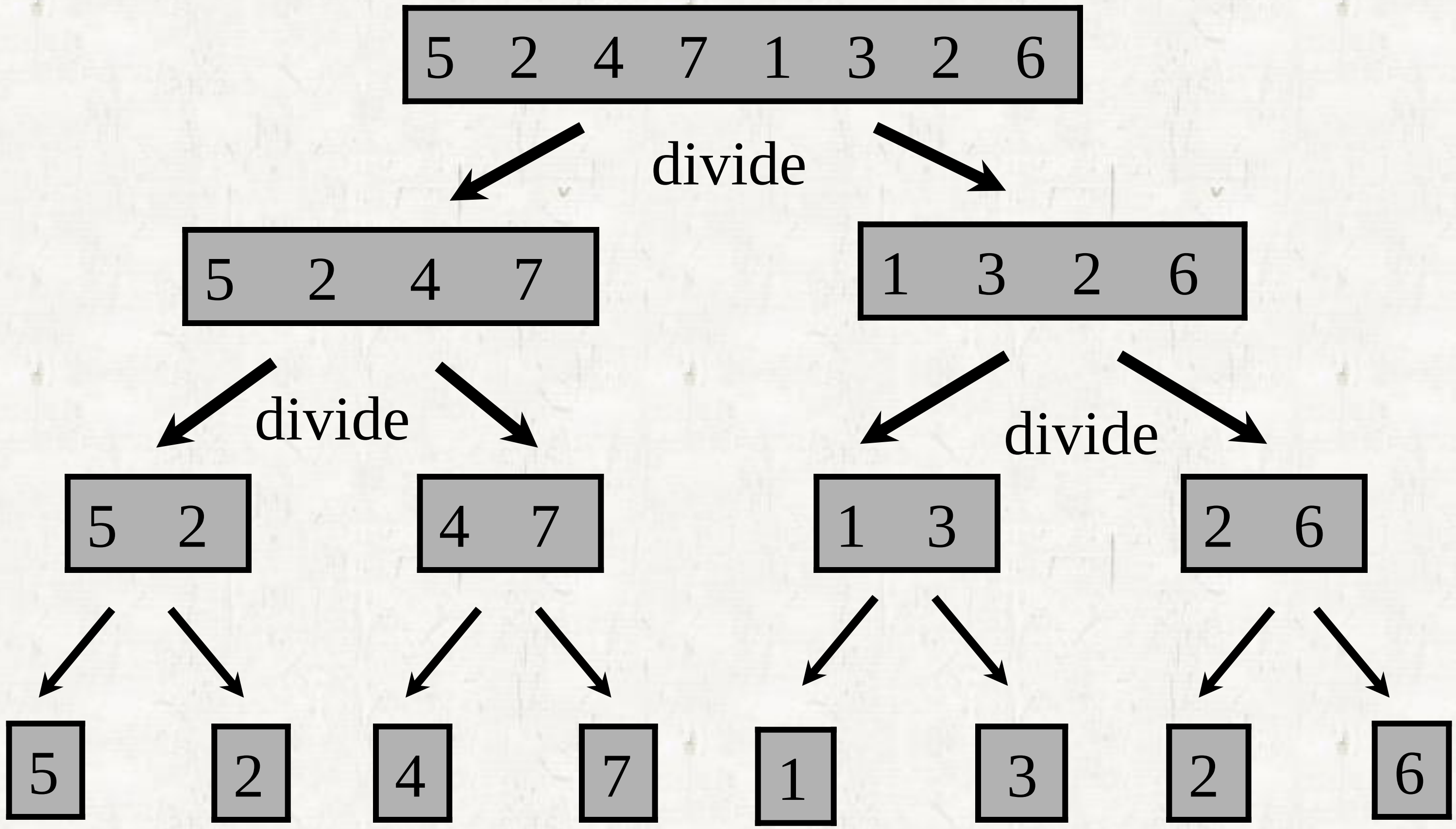
Merge

- Running time of merge
  - Let n_1 and n_2 denote the lengths of two sorted lists.
  - $\Theta(n_1 + n_2)$ time.
    - Main operations: compare and move
    - #comparison $\leq$ #movement
    - Obviously, #movement $= n_1 + n_2$
    - So, #comparison $\leq n_1 + n_2$
    - Hence, #comparison + #movement $\leq 2(n_1 + n_2)$
    - which means $\Theta(n_1 + n_2)$.

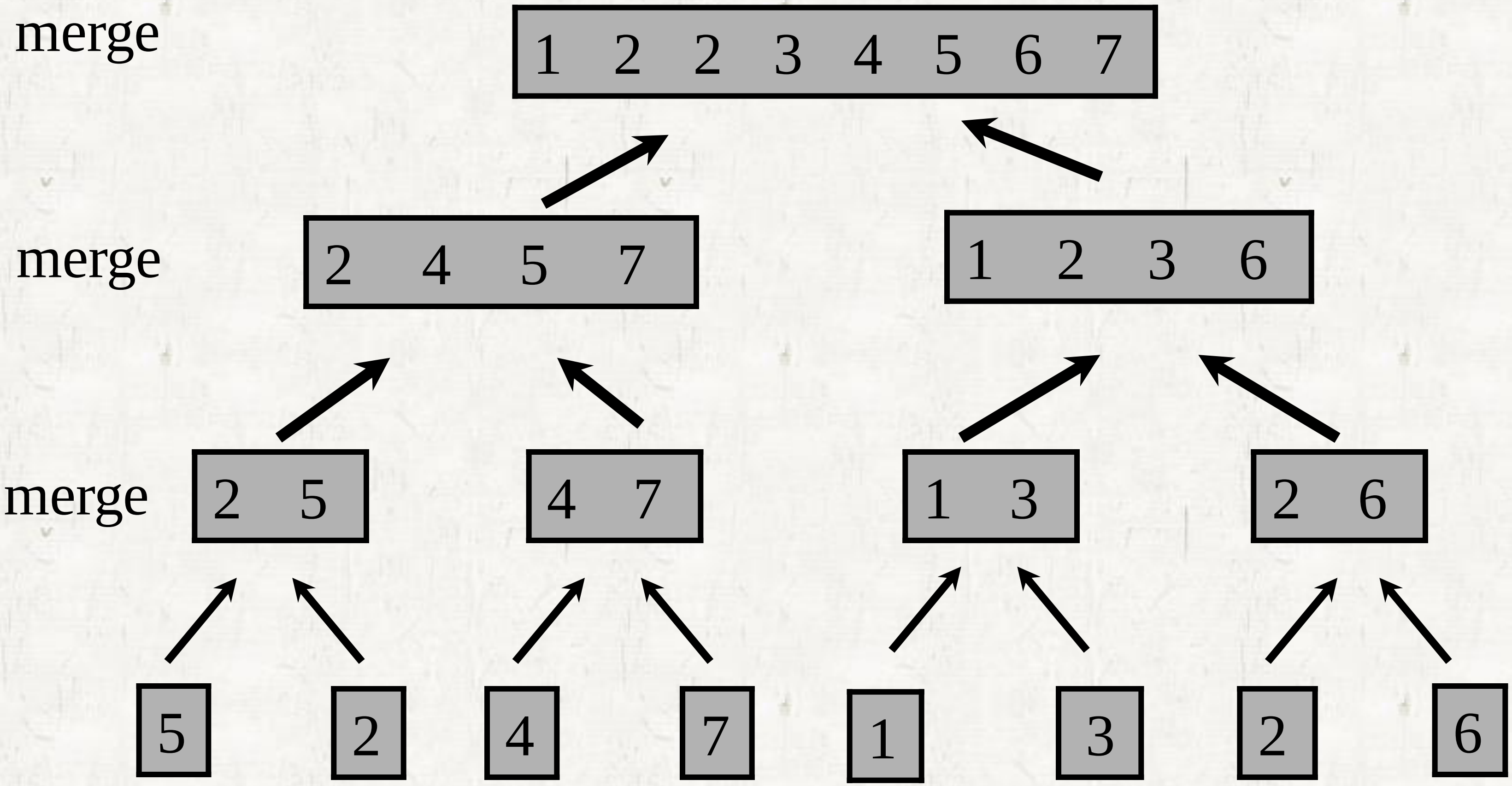
Merge sort

- A divide-and-conquer approach
  - **Divide:** Divide the *n* keys into two lists of *n*/2 keys.
  - **Conquer:** Sort the two lists recursively using merge sort.
  - **Combine:** Merge the two sorted lists.

Merge sort


5 2 4 7 1 3 2 6
divide
5 2 4 7
1 3 2 6
divide
divide
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6

Merge sort



Pseudo code

MERGE-SORT(A, p, r)

if $p < r$

  $q = \lfloor (p + r)/2 \rfloor$

  MERGE-SORT(A, p, q)

  MERGE-SORT(A, $q + 1$, r)

  MERGE(A, p, q, r)

Running time

- **Divide:** $\Theta(1)$
  - The divide step just computes the middle of the subarray, which takes constant time.
- **Conquer:** $2T(n/2)$
  - We recursively solve two subproblems, each of size $n/2$.
- **Combine:** $\Theta(n)$
  - We already showed that merging two sorted lists of size $n/2$ takes $\Theta(n)$ time.

Running time

- $T(n)$ can be represented as a recurrence.

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

Running time

- where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.

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

$$\Downarrow$$

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

Recursion tree

$T(n)$

cn

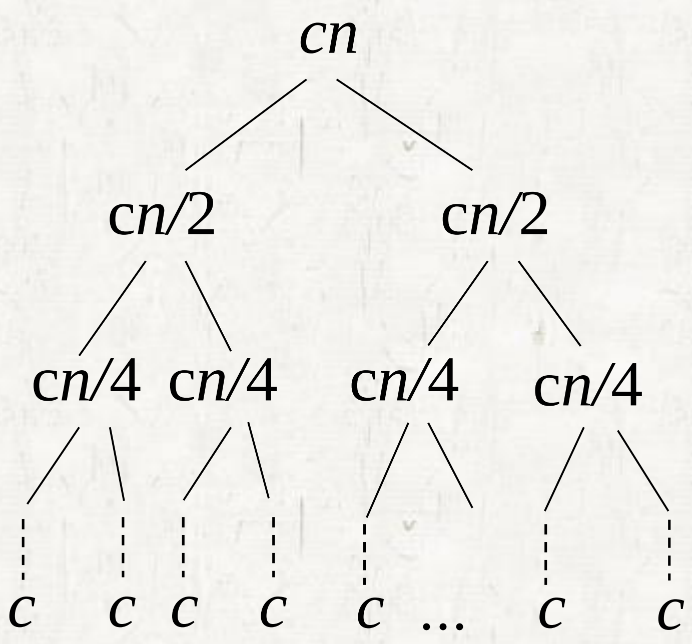
$T(n/2)$ $T(n/2)$

cn

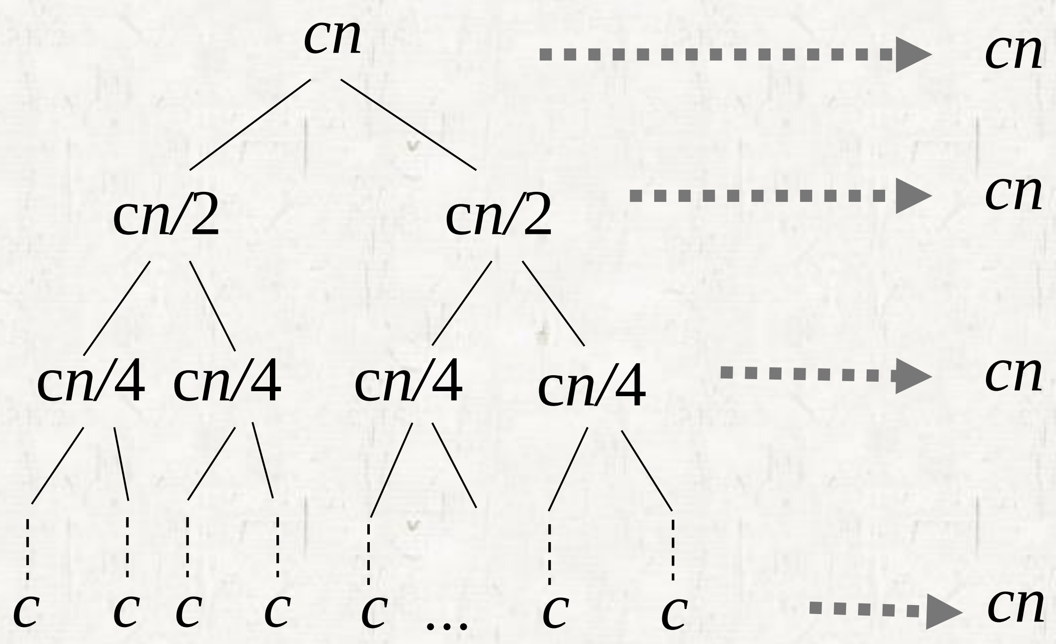
$cn/2$ $cn/2$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

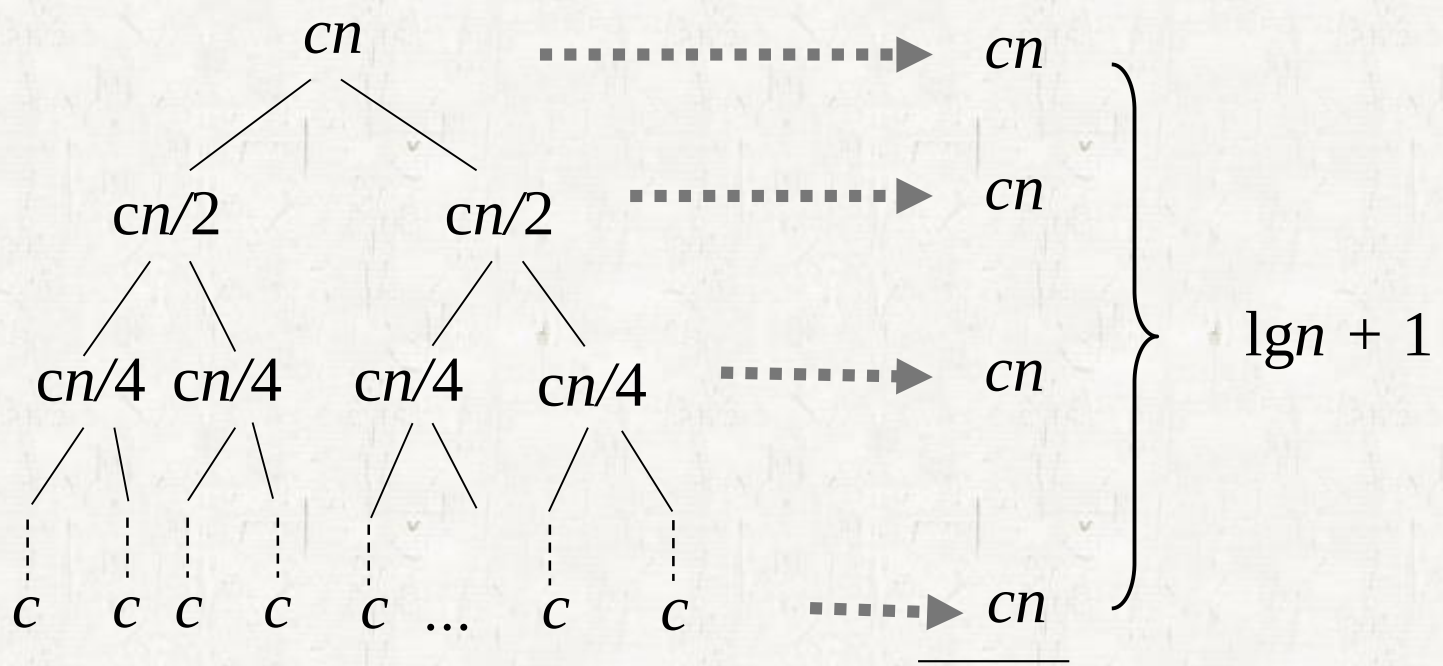
Recursion tree



Recursion tree



Recursion tree



Total : $cn\lg n + cn = \Theta(n\lg n)$

Self-study

- **Merge sort**
  - Exercise 2.3-1
  - Exercise 2.3-2
- **Horner's rule**
  - Problem 2-3 (a) (b)

More (sorting) algorithms

- **Binary Search**
  - Exercise 2.3-5

- **Selection sort**
  - Exercise 2.2-2