Design and Analysis of Algorithms

Getting started

Reference:
CLRS Chapter 2

Topics:

- **the basic concepts**
- **asymptotic analysis**

Algorithms

- **Algorithm.**
  - **A well-defined computational procedure that takes some value,or set of values, as input and produces some value,or set of values, as output.**

  - **issues: correctness, efficiency(amount of work done and space used), storage(simplicity,clarity), optimality .etc.**

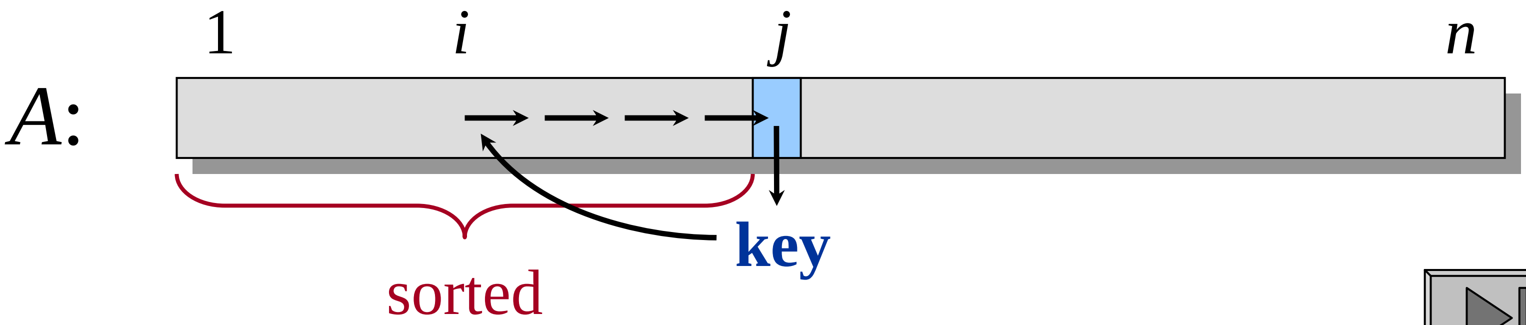
The problem of sorting

- Input: sequence $<a_1, a_2, \ldots, a_n>$ of n natural numbers
- Output: permutation $<a'_1, a'_2, \ldots, a'_n>$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- Example
  - Input: <5, 2, 4, 6, 1, 3>
  - Output: <1, 2, 3, 4, 5, 6>

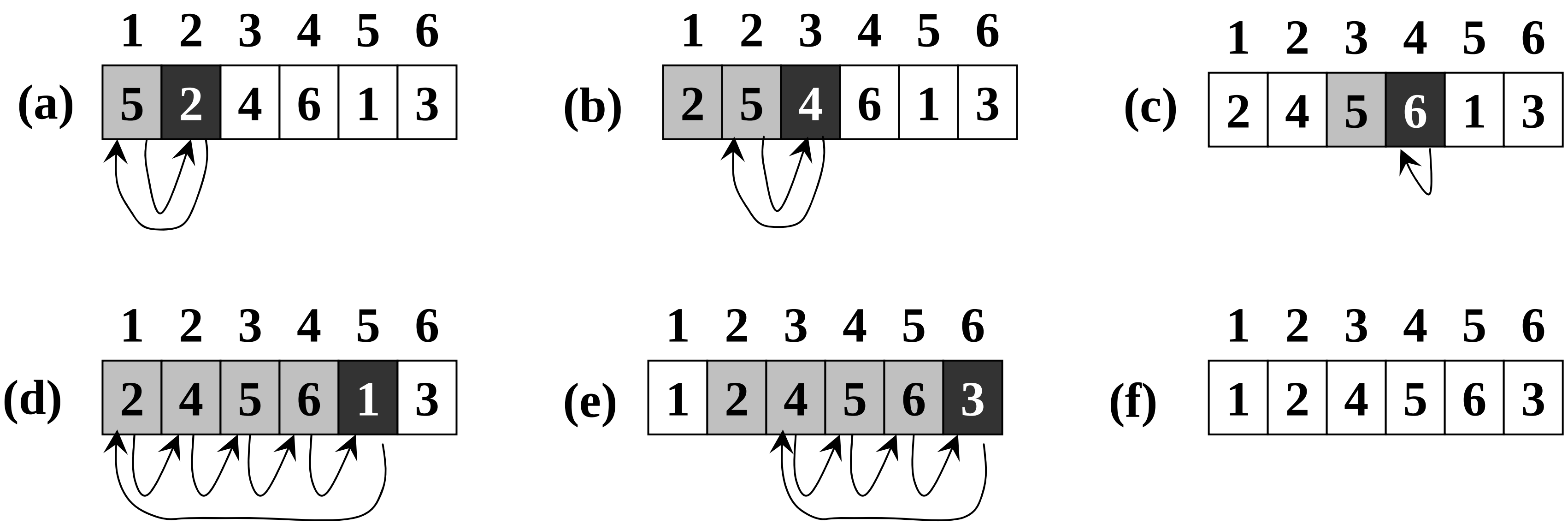
Insertion Sort

INSERTION SORT

```
INSERTION-SORT(A)
for j ← 2 to length(A)
    do key ← A[j]
          // insert A[j] into the sorted sequence A[1..j-1]
       i ← j - 1
       while i > 0 and A[i] > key
          do A[i+1] ← A[i]    // move item back
             i ← i - 1
       A[i+1] ← key    //find the insertion position
```



Insertion Sort Example



- **The operation of INSERTION-SORT on the array A = < 5, 2, 4, 6, 1, 3 >.**

Analysis of Insertion sort

INSERTION SORT			
	INSERTION-SORT(A)	cost	times
1	**for** j ← 2 **to** length(A)	c_1	n
2	**do** 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	**do** 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	$n-1$

t_j : the number of times the while loop test in line 5 is executed for that value of j

Analysis of Insertion sort

- To compute $T(n)$, the running time of Insertion-sort, we sum the products of the cost and times columns, obtaining

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

- The best-case occurs if the array is already sorted.

$$T(n)=c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5 (n\text{-}1) + c_8(n\text{-}1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) n\text{-} (c_2 + c_4 + c_5 + c_8)$$

  – The running time is a linear function of n

Analysis of Insertion sort

- The **worst-case** results if the array is in reverse sorted order – that is, in decreasing order.

$$T(n)= c_1 n + c_2(n-1) + c_4(n-1) + c_5 (n(n+1)/2-1) + c_6(n(n-1)/2)$$
$$+ c_7 (n(n-1)/2) + c_8(n-1)$$
$$= (c_5/2 + c_6/2 + c_7/2)n^2$$
$$+ (c_1 + c_2 + c_4 + c_5/2 – c_6/2 – c_7/2 + c_8) n – (c_2 + c_4 + c_5 + c_8)$$

  – The running time is a quadratic function of n

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

$$\sum_{j=2}^{n} t_j -1 = \sum_{j=2}^{n} (j–1) = n(n-1)/2$$

Worst-case and Average-case Analysis

- **Note:**
  - Upper bound on the running time for any input
  - For some algorithms, worst-case occur fairly often.
    - » e.g. Searching in a database for a particular piece of information
  - Average case often as bad as worst case (but not always!)

Order of Growth

- **We will only consider order of growth of running time:**
  - **We can ignore the lower-order terms, since they are relatively insignificant for very large *n*.**
  - **We can also ignore leading term's constant coefficients, since they are not as important for the rate of growth in computational efficiency for very large *n*.**
  - **We just said that best case was linear in *n* and worst/average case quadratic in *n*.**

Designing Algorithms

- We discussed insertion sort
  - We introduced RAM model of computation
  - We analyzed insertion sort in the RAM model
  - We discussed how we are normally only interested in growth of running time:
    - » Best-case linear in $O(n)$, worst-case quadratic in $O(n^2)$
- Can we design better than n^2 sorting algorithm?
- We will do so using one of the most powerful algorithm design techniques.

Divide-and-Conquer

- Recursive in structure

- To solve **P**:
  - Divide **P** into smaller problems $P_1, P_2, \ldots, P_k$.
  - Conquer by solving the (smaller) subproblems recursively.
  - Combine the solutions to $P_1, P_2, \ldots, P_k$ into the solution for **P**.

Merge Sort Algorithm

- **Using divide-and-conquer, we can obtain a merge-sort algorithm**
  - **Divide: Divide the n elements into two subsequences of $n/2$ elements each.**
  - **Conquer: Sort the two subsequences recursively.**
  - **Combine: Merge the two sorted subsequences to produce the sorted answer.**

- **Assume we have procedure MERGE(A, p, q, r) which merges sorted $A[p...q]$ with sorted $A[q+1..r]$ in ($r - p$) time.**

Merge-Sort (A, p, r)

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

OUTPUT: an ordered sequence of *n* numbers

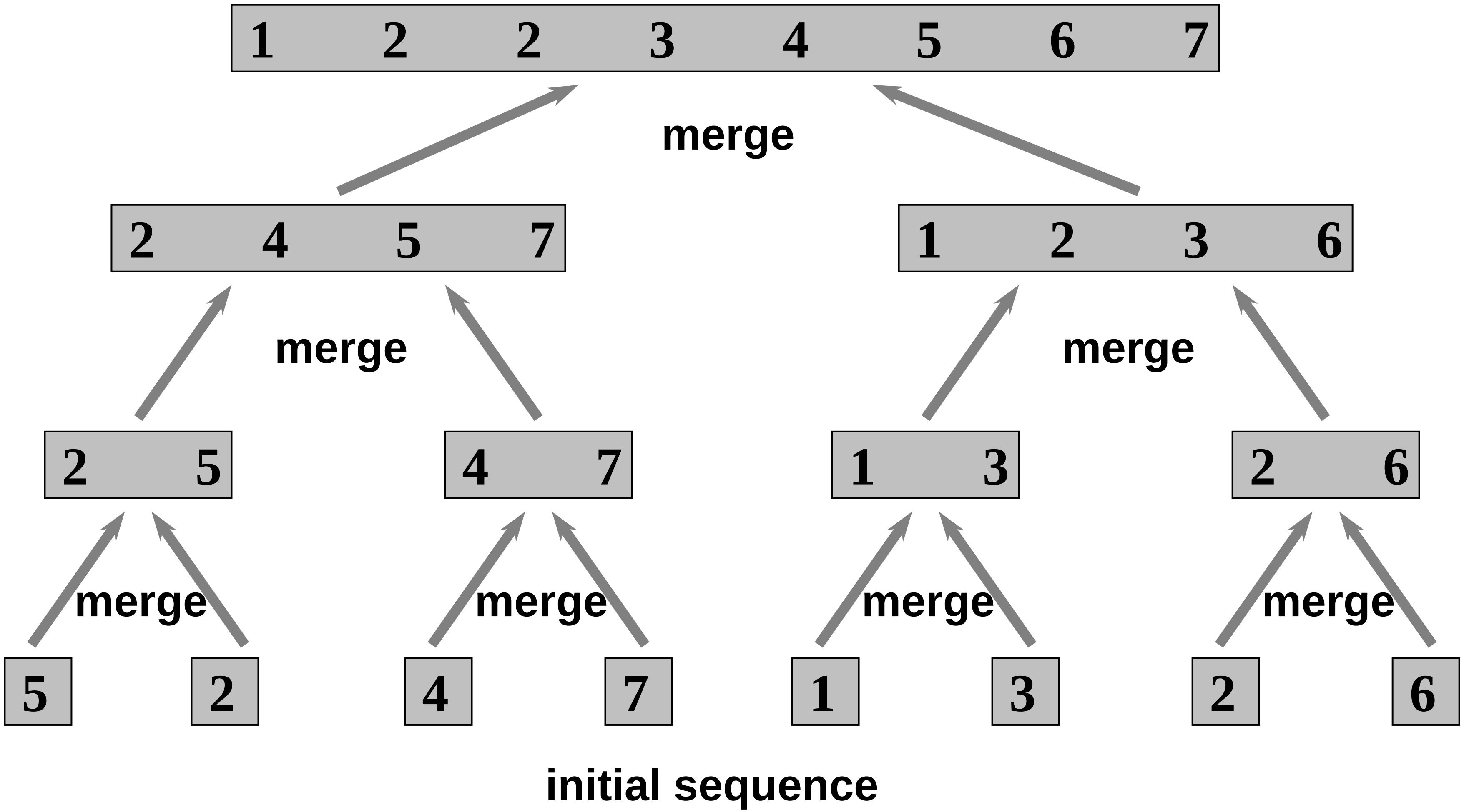
MERGESORT

```
MERGE-SORT(A,p,r)
if  p < r
    then  q ← ⌊(p+r)/2⌋
          MERGE-SORT (A, p, q)
          MERGE-SORT (A, q+1, r)
          MERGE (A, p, q, r)
```

MERGE

```
MERGE(A, p, q, r)
n1 ← q-p+1;
n2 ← r-q;
create arrays L[1..n1+1] and R[1..n2+1]
 for i ← 1 to n1
     do L[i] ← A[p + i-1]
 for j ← 1 to n2
     do R[j] ← A[q + j]
 L[n1+1] ← ∞
 R[n2+1] ← ∞      //set sentinel
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
                j ← j + 1
```

Action of Merge Sort



Analysis divide-and-conquer algorithms

- Let $T(n)$ be the running time on a problem of size n.
  - Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original.
  - $D(n)$ the time to divide the problem into subproblems
  - $C(n)$ the time to combine the solutions to subproblems into the solution to the original problem

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

Recurrence Equation

Mergesort Analysis

- How long does mergesort take?
  - Bottleneck = merging (and copying).
    - » merging two files of size $n/2$ requires n comparisons
  - $T(n)$ = comparisons to mergesort n elements.
    - » to make analysis cleaner, assume n is a power of 2

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Claim. $T(n) = n \lg_2 n$.
  - Note: same number of comparisons for ANY file.
    - » even already sorted
  - We'll prove several different ways to illustrate standard techniques.

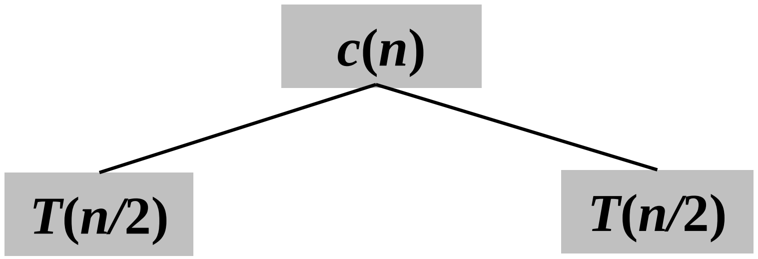
Proof by Picture of Recursion Tree

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$

$T(n)$

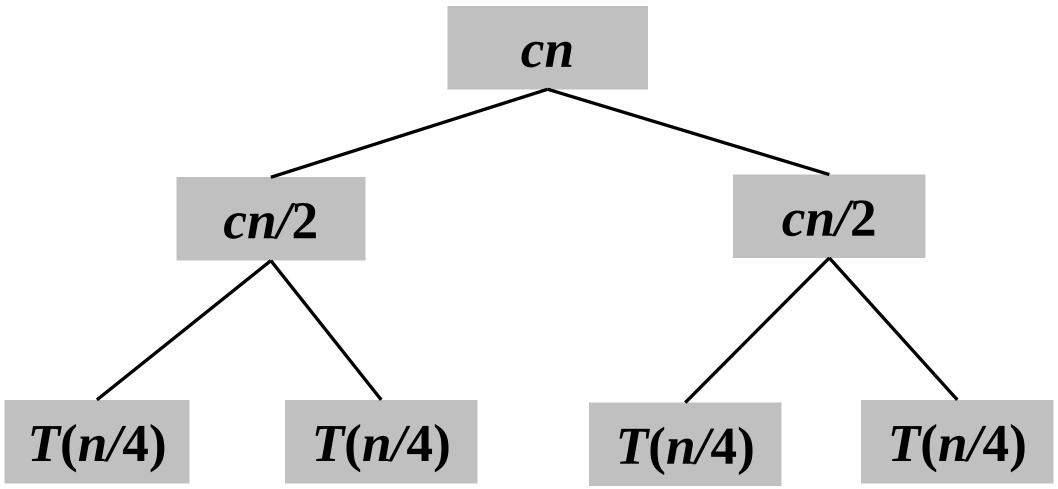
Proof by Picture of Recursion Tree

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$

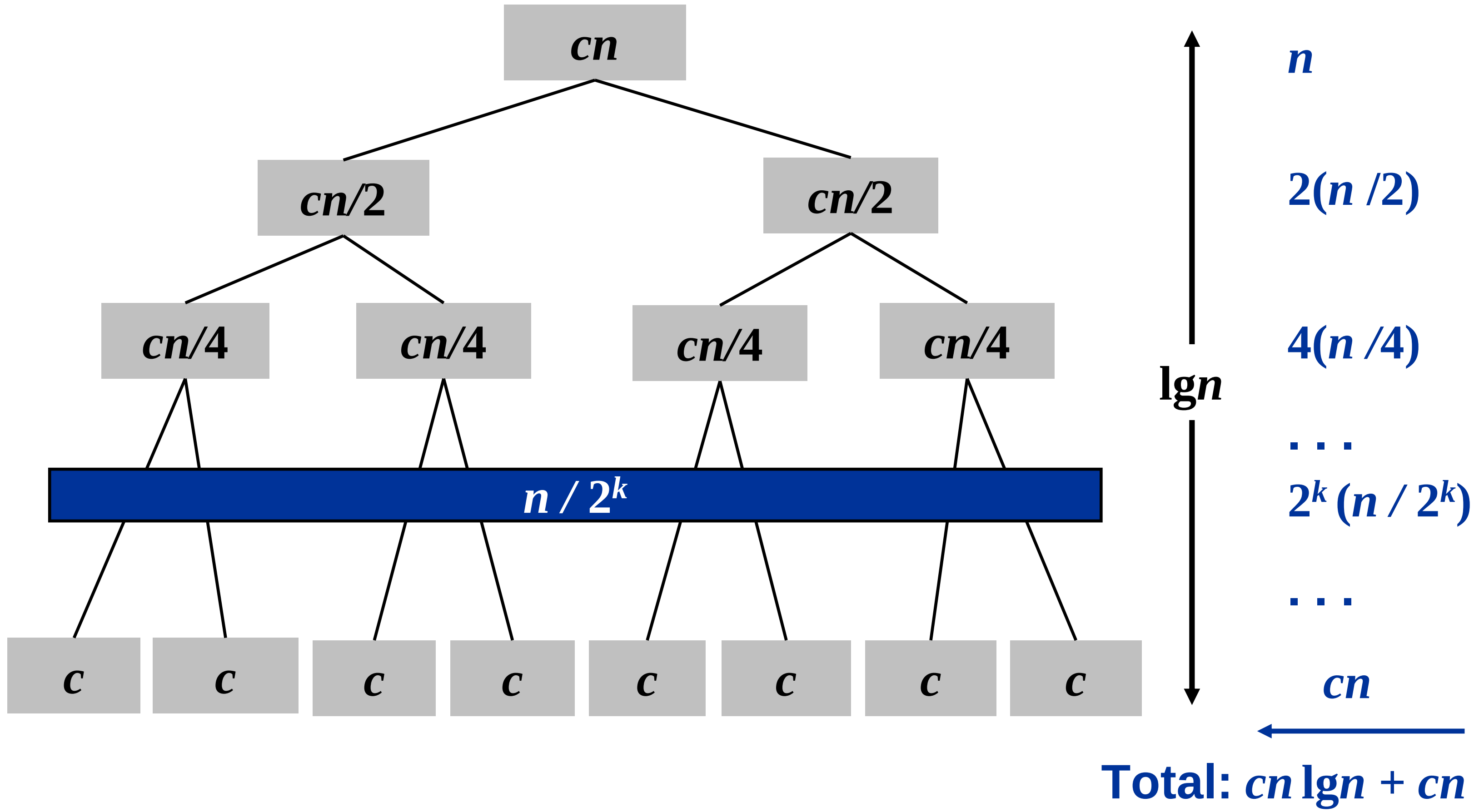


Proof by Picture of Recursion Tree

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



Construction of recursion tree



The fully expanded tree has $\lg n + 1$ levels, i.e., it has height $\lg n$, and each level contributes a total cost of cn. The total cost is $\Theta(n \lg n)$.