Divide and Conquer (Merge Sort)

Divide and Conquer

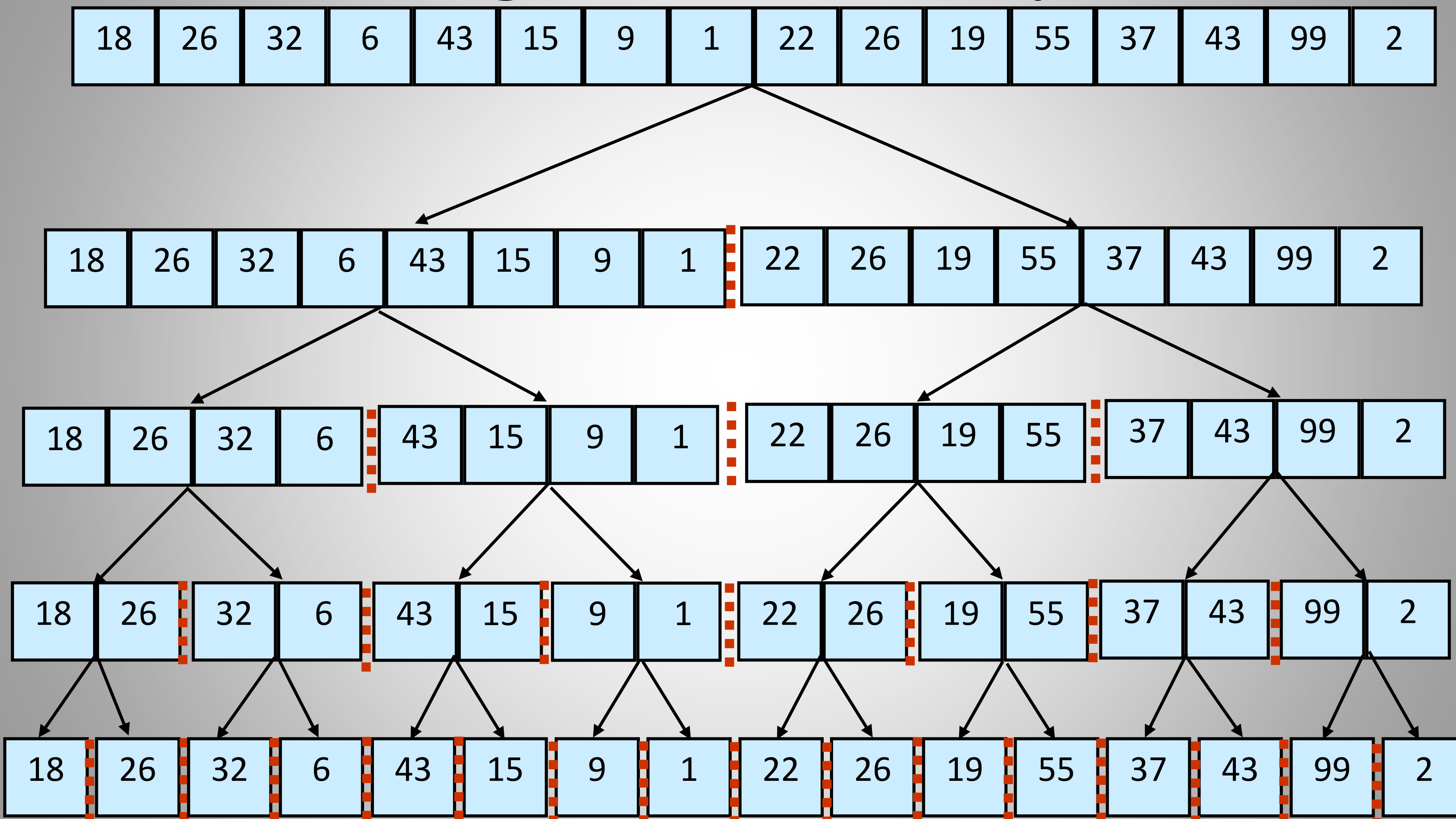
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

An Example: Merge Sort

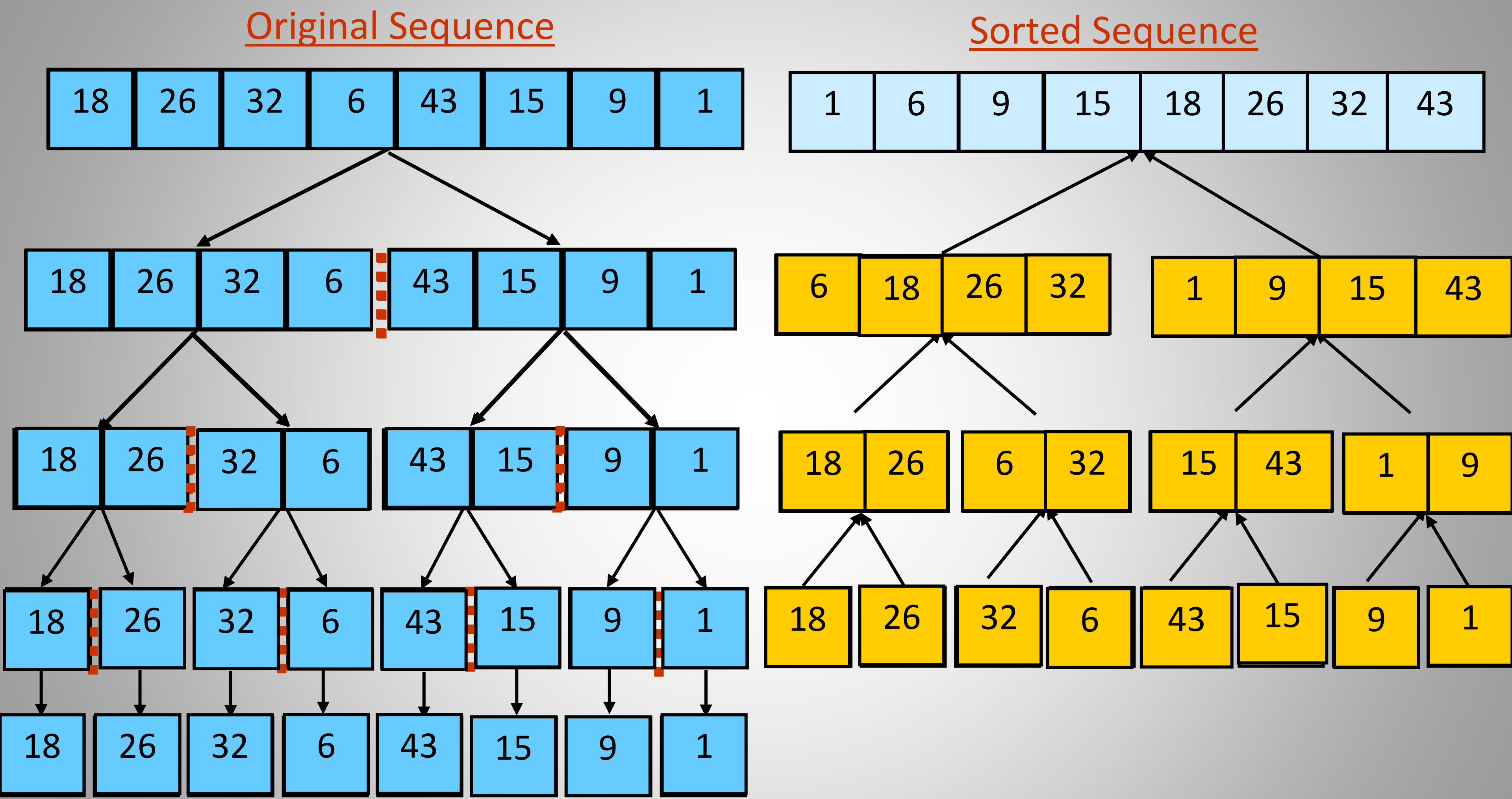
***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***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 – 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
    if $p < r$
        then $q \leftarrow \lfloor (p+r)/2 \rfloor$
            MergeSort (A, p, q)
            MergeSort $(A, q+1, r)$
            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*)

```
n1 ← q − p + 1
n2 ← r − q
    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] ← ∞
    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
```

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

A

…	1	6	8	9	26	32	42	43	…

k

L

6	8	26	32	∞

i

R

1	9	42	43	∞

j

Correctness of Merge

```
Merge(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
    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] ← ∞
    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
```

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.

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and takes at most cn steps, for some constant c.
- Likewise, the basis case ($n < 2$) will take at c most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

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

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Recurrence Relations

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- **Solution Methods**
  - Substitution Method.
  - Iteration Method.
  - Recursion-tree Method.
  - Master Method.
- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.

Iterative Method

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/2^2)) + c(n/2)) + cn \\
&= 2^2 T(n/2^2) + 2cn \\
&= 2^3 T(n/2^3) + 3cn \\
&= 2^4 T(n/2^4) + 4cn \\
&= \ldots \\
&= 2^i T(n/2^i) + icn
\end{aligned}
$$

- Note that base, T(n)=c, case occurs when 2^i=n. That is, i = log n.
- So,

$$T(n) = cn + cn\log n$$

- Thus, T(n) is O(n log 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:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$

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

- Rewrite the recurrence as

$$T(n) = \boldsymbol{c} \qquad \text{if } n = 1$$

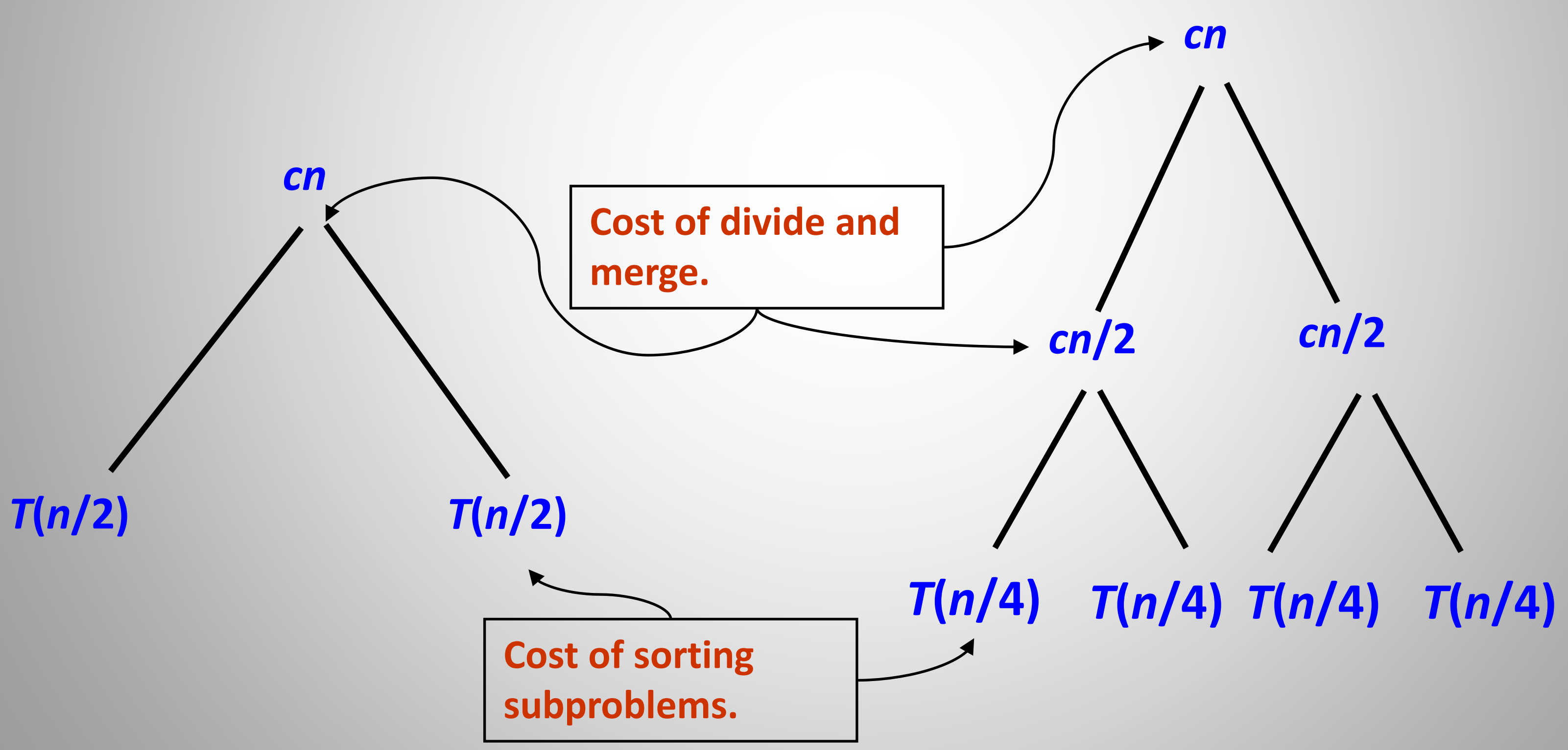
$$T(n) = 2T(n/2) + \boldsymbol{cn} \qquad \text{if } n > 1$$

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

Recursion Tree for Merge Sort

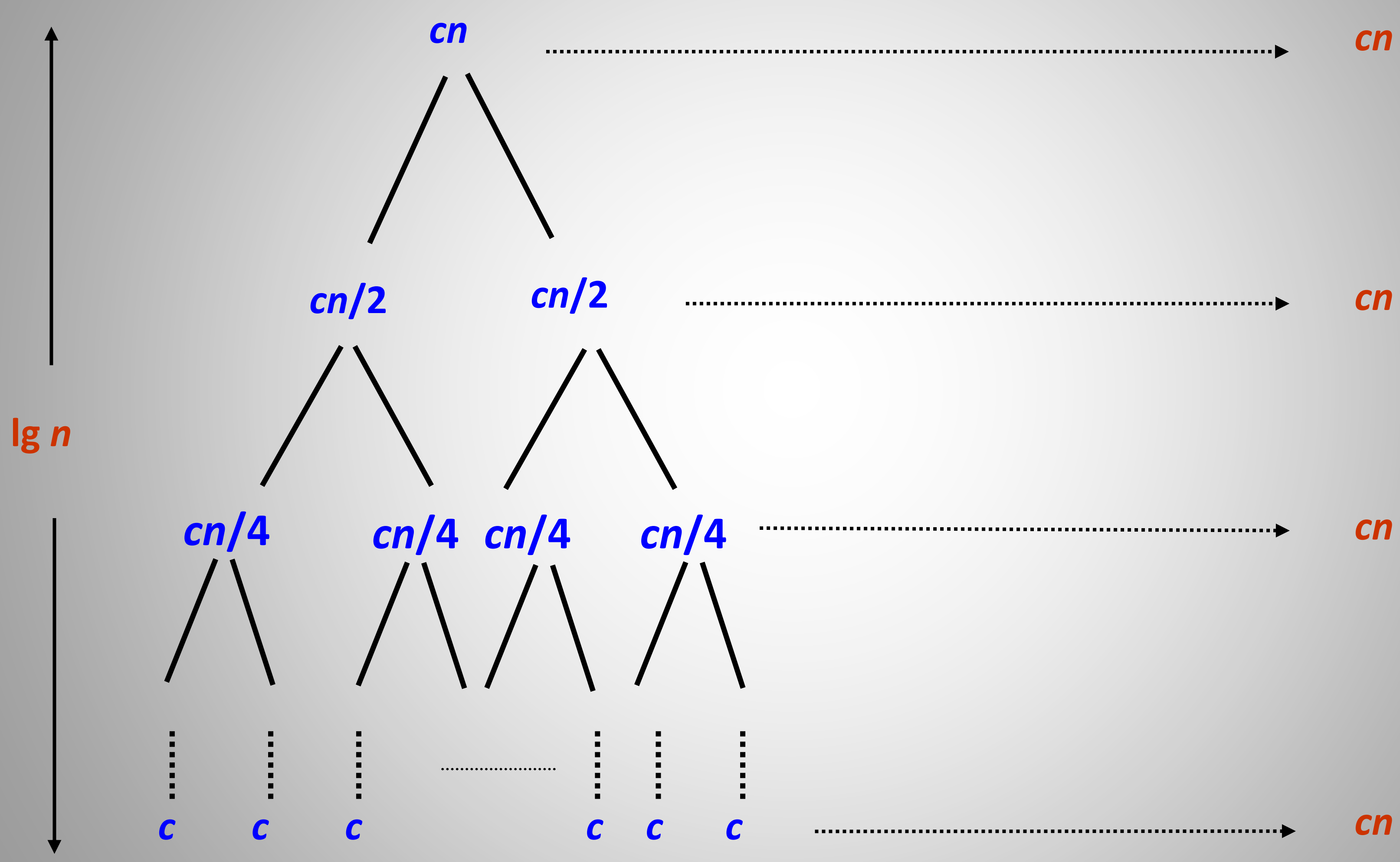
For the original problem, we have a cost of *cn*, plus two subproblems each of size (n/2) and running time $T(n/2)$.

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

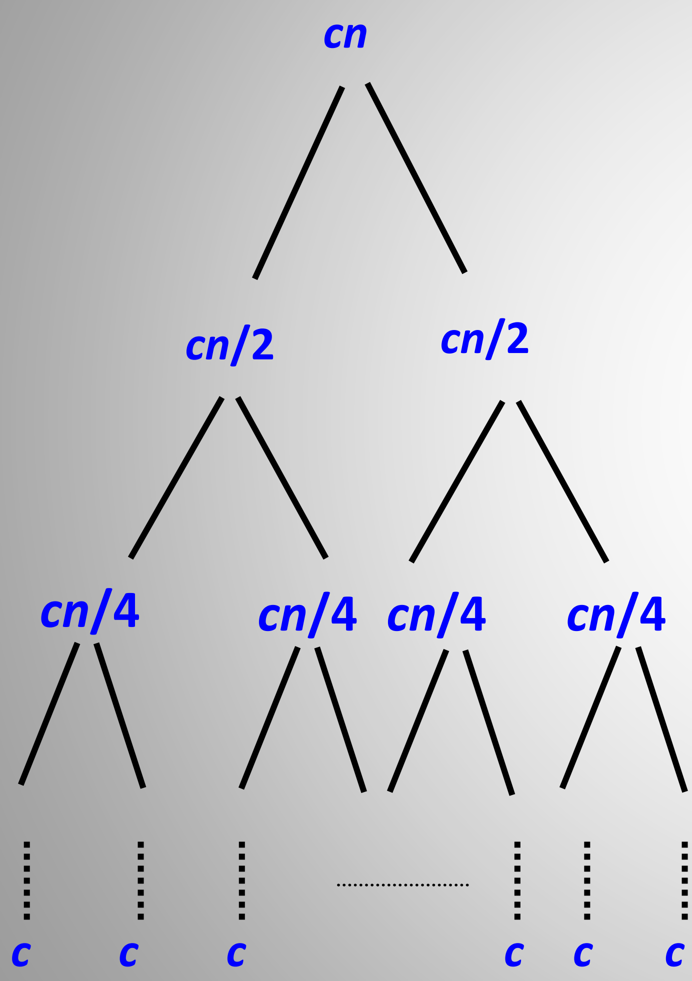
Continue expanding until the problem size reduces to 1.



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 $\Rightarrow$ *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* = Θ(*n* lg*n*).

Master Method

- Recurrence: Relation for merge Sort

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

a= 2 b=2 f(n)= cn

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

$T(n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n^1 \lg n) = \Theta(n \lg n)$

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.

The substitution method

1. Guess a solution
2. Use induction to prove that the solution works

Substitution method

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \le d\ g(n)$, for some $d > 0$ and $n \ge n_0$ (strong induction)
  - Induction hypothesis: $T(k) \le d\ g(k)$ for all $k < n$
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants d and n_0 for which the **induction goal** holds

The Substitution method

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

- **Guess**: $T(n) = O(n \log n)$
- **Proof** by <u>Mathematical Induction</u>:

Prove that $T(n) \leq d\, n \log n$ for $d>0$

$T(n) \leq 2(d \cdot n/2 \cdot \log n/2) + cn$

(where $T(n/2) \leq d \cdot n/2 (\log n/2)$ *by induction hypothesis)*

$\leq dn \log n/2 + cn$

$= dn \log n - dn + cn$

$= dn \log n + (c-d)n$

$\leq dn \log n$ if $d \geq c$

- Therefore, ***$T(n) = O(n \log n)$***

Substitution Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \ge 2 \end{cases}$$

- Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than cn log n

Substitution Method, Part 2

- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: T(n) < cn $\log^2$ n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \end{aligned}$$

  - if c > b.
- So, T(n) is O(n $\log^2$ n).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.