# Divide and Conquer Algorithm

# Divide and Conquer

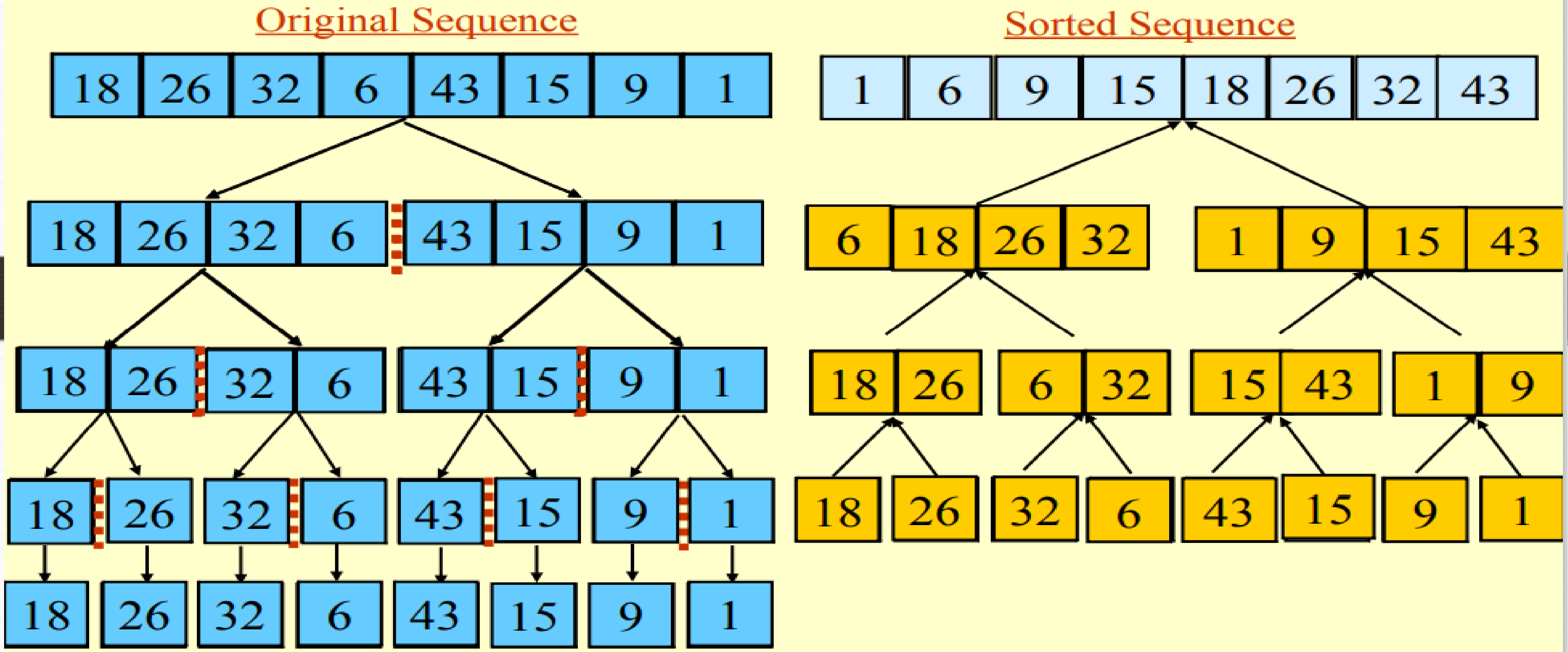
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create 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 (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 ← ⌊(p+r)/2⌋
            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.

# Analysis of Merge Sort

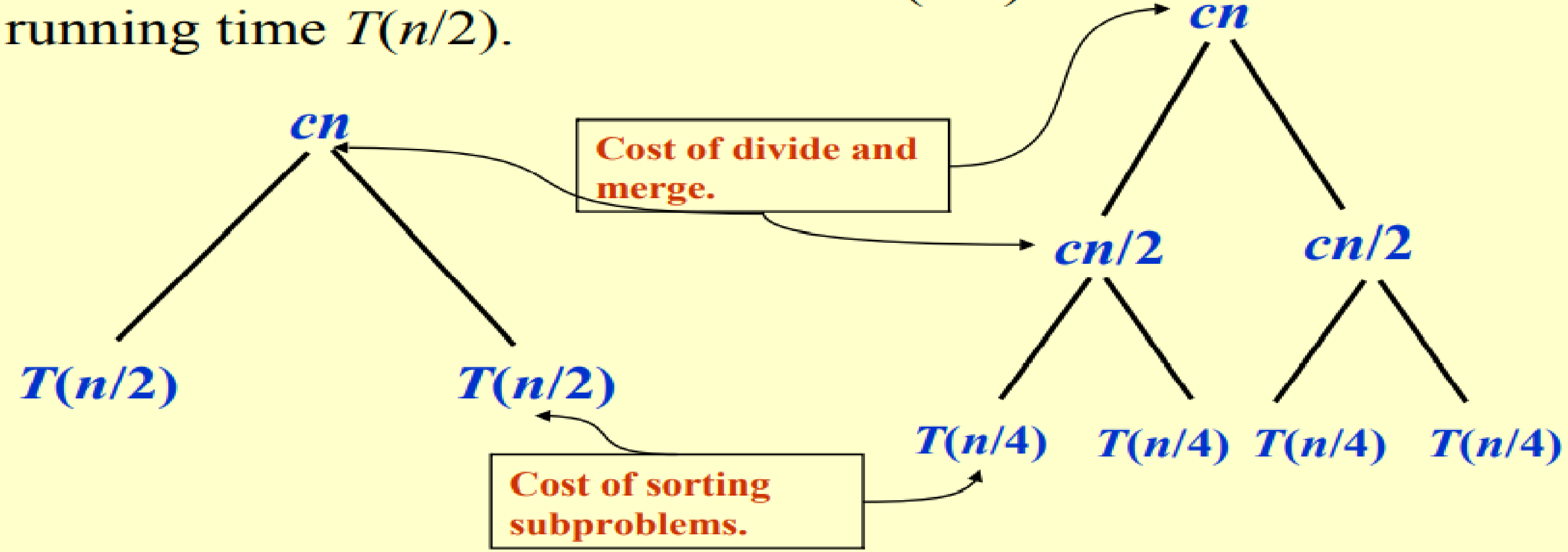
- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging $n$ elements takes $\Theta(n)$
- Total:

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

# Recursion Tree for Merge Sort

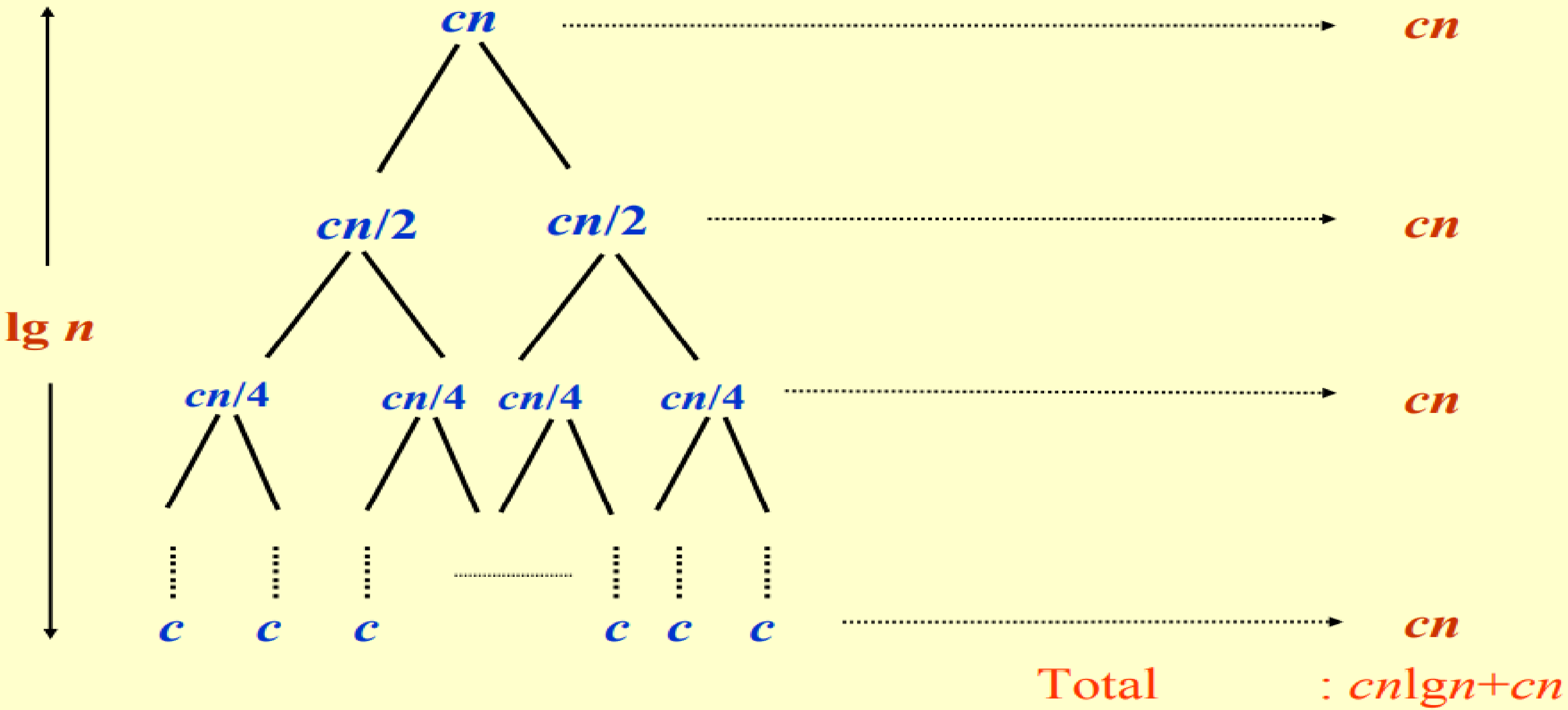
For the original problem, we have a cost of $cn$, plus two subproblems each of size $(n/2)$ and running time $T(n/2)$.

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



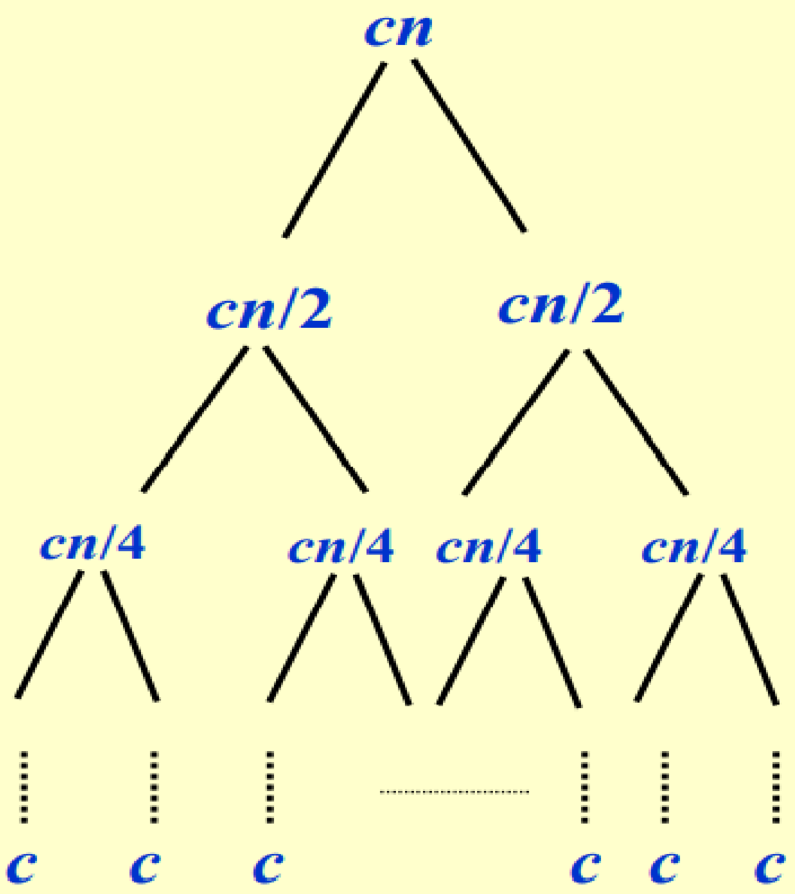
# Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



# 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 = $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$.

# Pros and Cons

- **Pros:**
- Large size list (Millions of records)
- Linked List
- External sorting
- Stable

**Cons:**

- Extra space (Not in place sort)
- No small problem