Divide and Conquer (Merge Sort)

- Divide and conquer
- Merge sort
- Loop-invariant
- Recurrence relations

Divide and Conquer

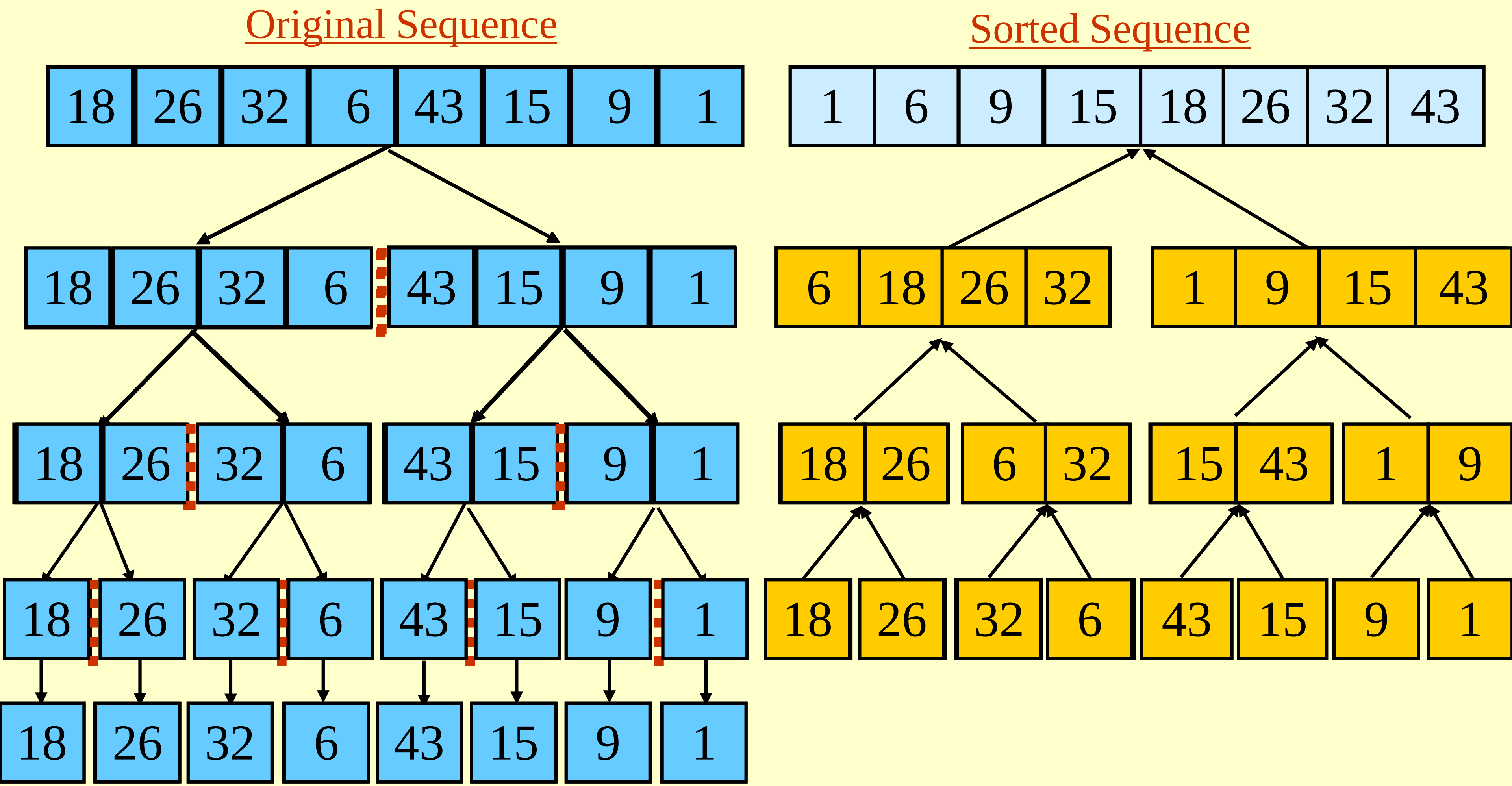
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions of the sub-problems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

1 **if** $p < r$

2   **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$

3   *MergeSort* (A, p, q)

4   *MergeSort* (A, q+1, r)

5   *Merge* (A, p, q, r) // merges $A[p..q]$ with $A[q+1..r]$

Initial Call: *MergeSort*(A, 1, n)

Procedure Merge

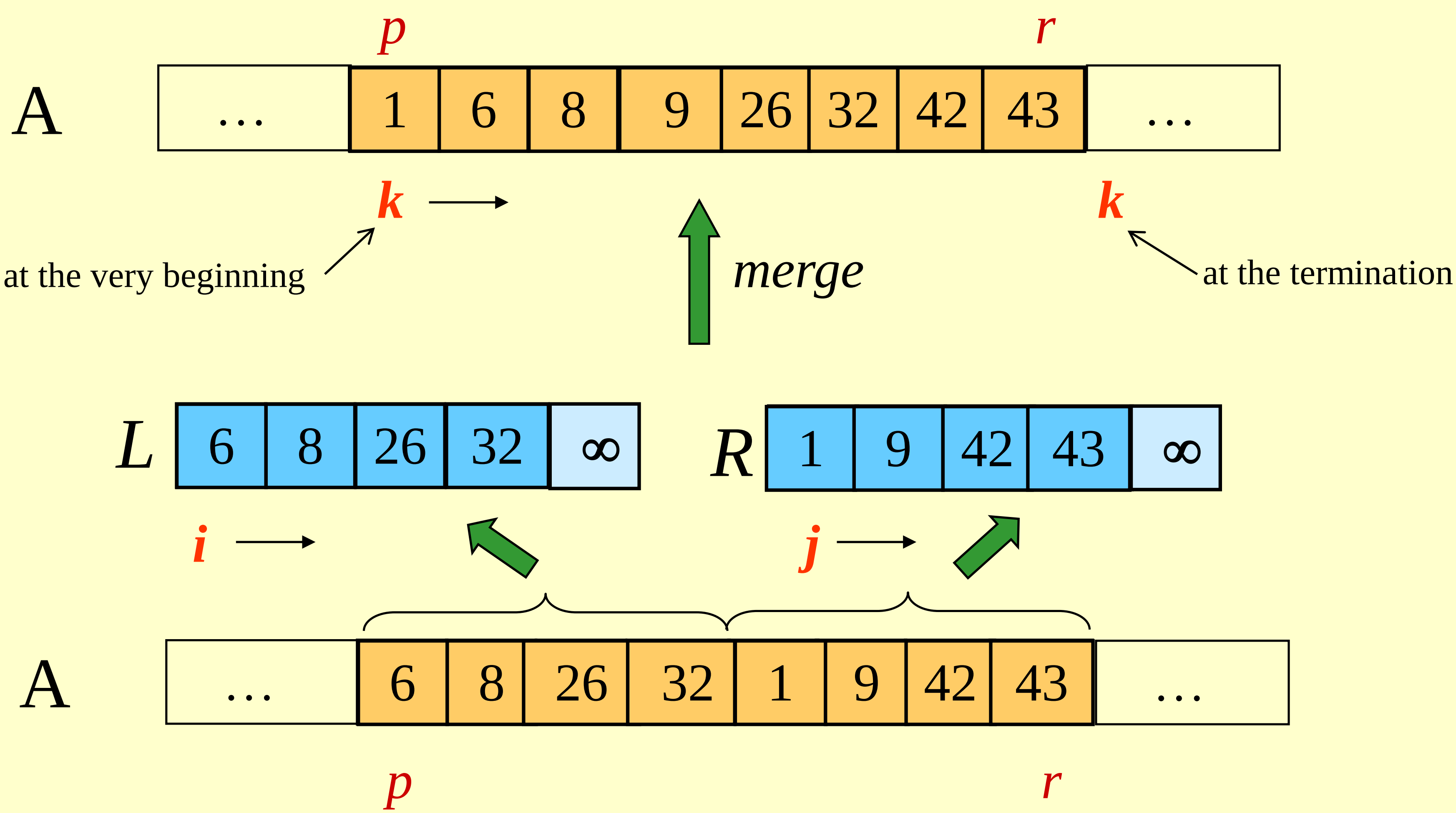
```
Merge(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
    for i ← 1 to n1
        do L[i] ← A[p + i – 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            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



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:

(We will prove that if after the *k*th iteration, the Loop Invariant (LI) holds, we still have the LI after the (*k*+1)th iteration.)

Case 1: $L[i] \le R[j]$

- By Loop Invariant, A contains $k - p$ smallest elements of L and R in *sorted order*.
- Also, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $k - p + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for Case 2: $L[i] > R[j]$.

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 Loop Invariant (LI), A contains $k - p$ 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 $k - p + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for Case 2: $L[i] > R[j]$.

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 $r - p + 3 - (r - p + 1) = 2$ elements. All but the two sentinels have been copied back into A.

Improvements

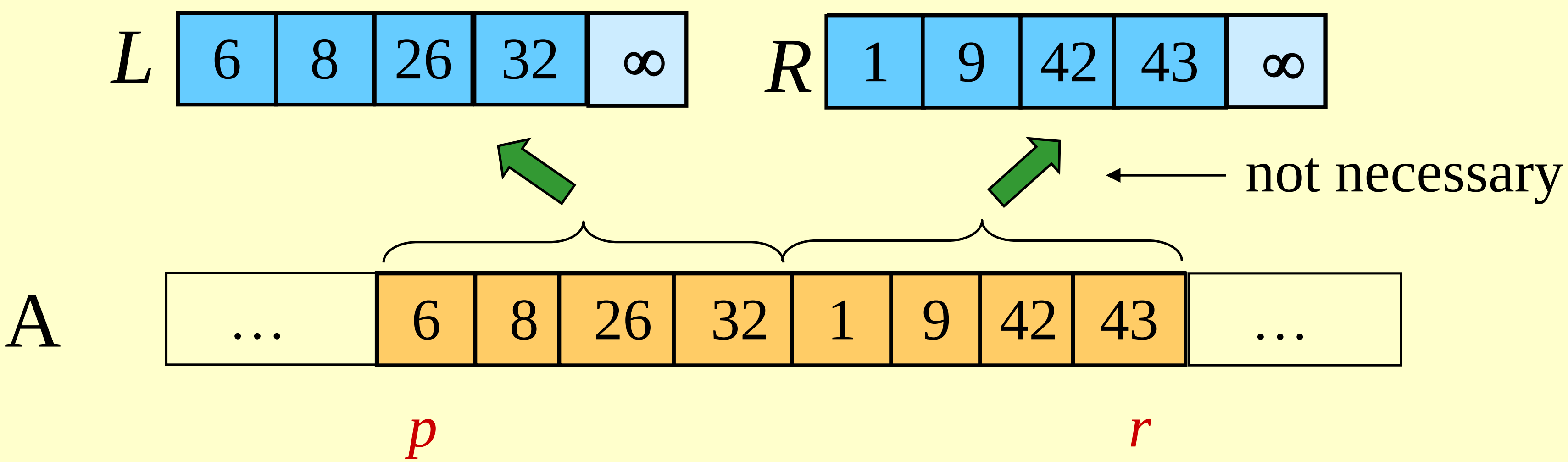
- Reduction of data movements
- Non-recursive Algorithm

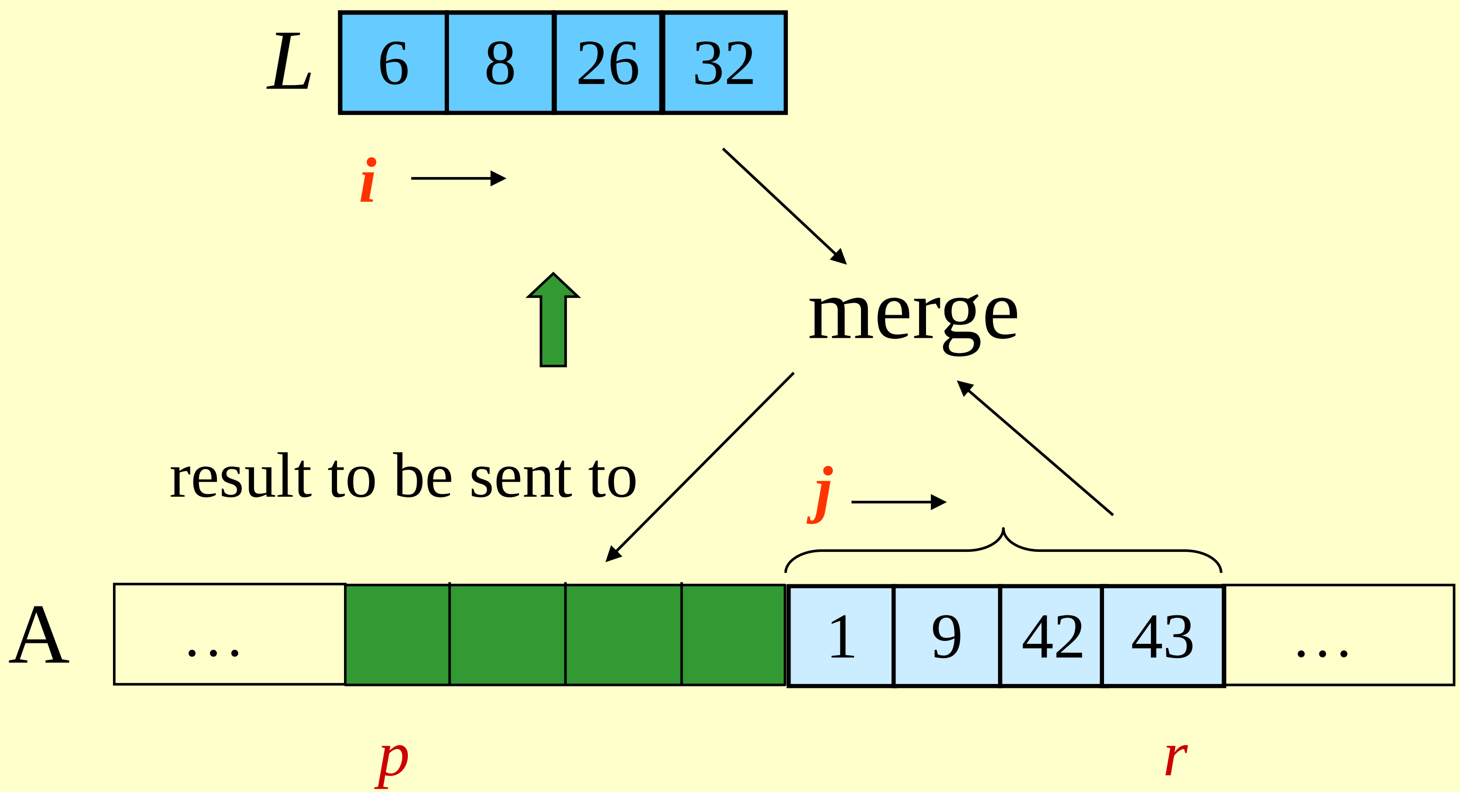
Y. Chen, and R. Su, Merge Sort Revisited, ACTA Scientific Computer Sciences, Vol. 4, No. 5, pp. 49 - 52, 2022.

Improvements

- Reduction of data movements

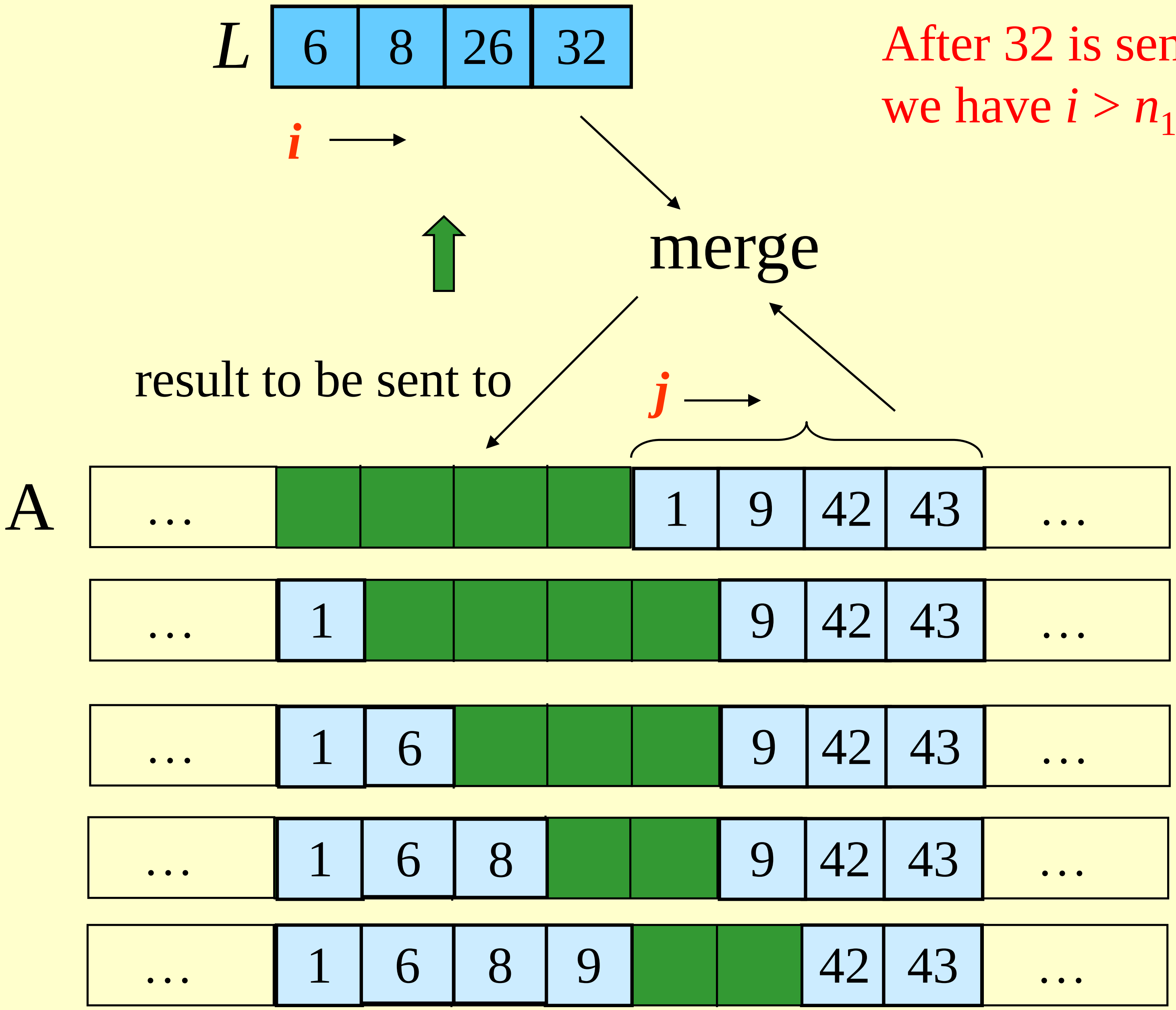
We notice that in the procedure $merge(\ )$ of Merge sort the copying of $A[q+1 \mathinner{..} r]$ into R is not necessary, since we can directly merge L and $A[q+1 \mathinner{..} r]$ and store the merged, but sorted sequence back into A.





L
6
8
26
32
i
merge
result to be sent to
j
A
…
1
9
42
43
…
p
r

After 32 is sent to A, we have $i > n_1$.



Why does it work?

- **Denote by A' the sorted version of A. Denote by $A'(i, j)$ a prefix of A' which contains the first i elements from L and first j elements from $A[q + 1 .. r]$.**

- **Obviously, we can store $A'(i, j)$ in A itself since after the jth element (from $A[q + 1 .. r]$) has been inserted into A', the first $q - p + j + 1$ entries in A are empty and $q - p + 1 \geq i$ (thus, $q - p + j + 1 \geq i + j$).**

Improvements

Algorithm: *mergeImpr* (A, p, q, r)
Input: Both $A[p .. q]$ and $A[q+1 .. r]$ are sorted; but A as a whole is not sorted
Output : sorted A

1. $n_1 := q - p + 1$; $n_2 := r - p + 1$; $k := p$;
2. let $L[1 .. n_1]$ be a new array;
3. **for** $i = 1$ to n_1 **do**
4. $\quad L[i] := A[p + i - 1]$
5. $\quad i := p$; $j := q + 1$;
6. **while** $i \leq n_1$ and $j \leq n_2$ **do**
7. $\quad$ **if** $L[i] \leq A[j]$ **then** $\{A[k] := L[i]; i := i + 1;\}$
8. $\quad$ **else** $\{A[k] := A[j]; j := j + 1;\}$
9. $\quad k := k + 1$;
10. **if** $j > n_2$ **then**
11. copy the remaining elements in L into $A[k .. r]$;

When going out of while-loop,
we distinguish between two cases:
$i > n_1$,
$j > n_2$.

Improvements

Non-recursive algorithm

•Merge Sort can be further improved by replacing its recursive calls with a series of merging operations, by which the recursive execution of the algorithm is simulated.

•The whole working process can be divided into $\lceil \log_2 n \rceil$ phases.

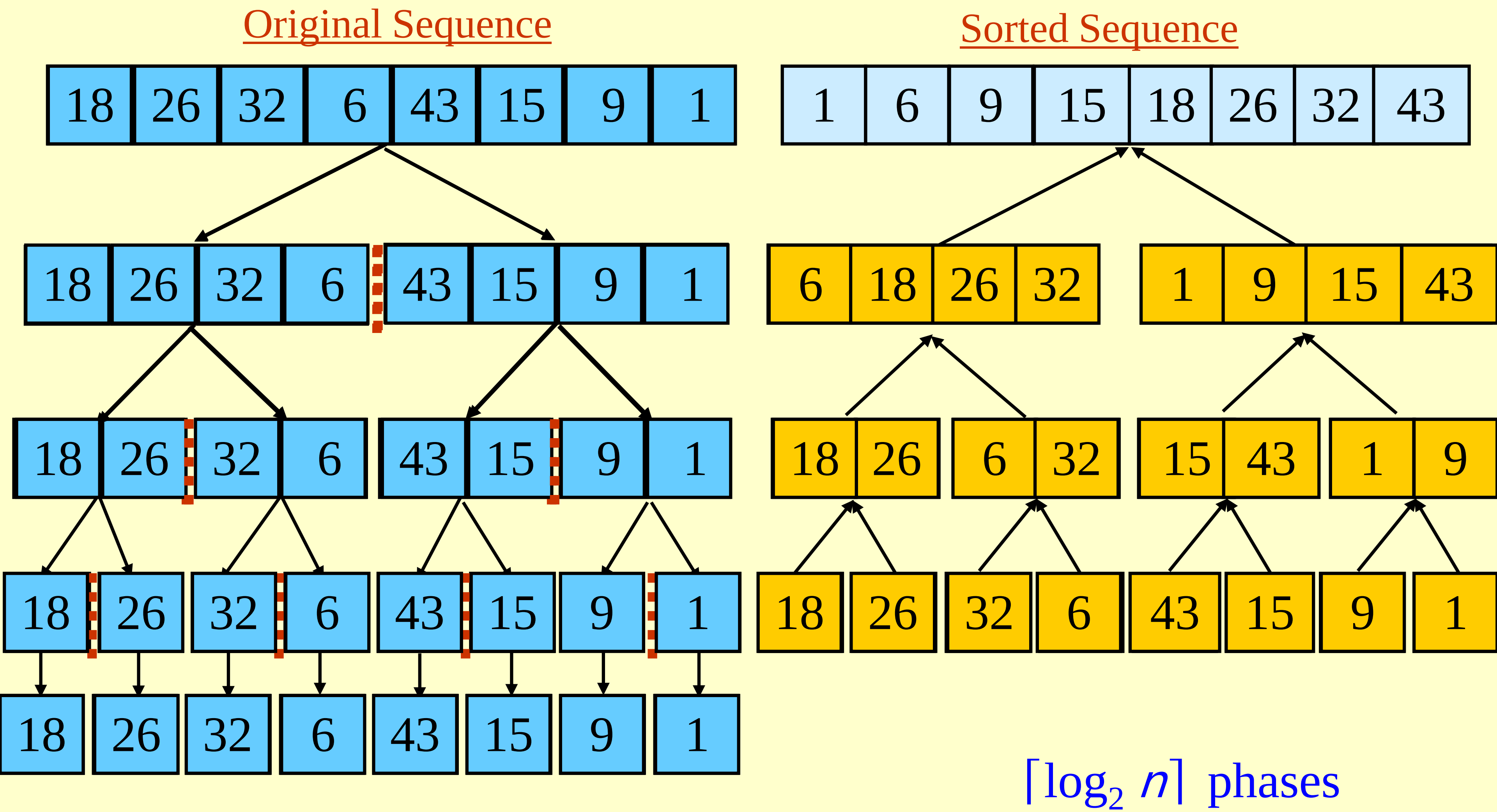
•In the first phase, we will make $\lceil n/2 \rceil$ merging operations with each merging two single-element sequences together.

•In the second phase, we will make $\lceil n/4 \rceil$ merging operations with each merging two two-element sequences together, and so on.

•Finally, we will make only one operation to merge two sorted subsequences to form a globally sorted sequence. Between the sorted subsequences, one contains $\lceil n/2 \rceil$ elements while the other contains $\lfloor n/2 \rfloor$ elements.

Improvements

Algorithm: $mSort$ (A)
Input : A - a sequence of elements stored as an array;
Output : sorted A
1. if $|A| \le 1$ **then** return A;
2. $r := |A|$;
3. $l := \lceil \log_2 r \rceil$;
4. $j := 2$;
5. for $i = 1$ to l **do**
6. **for** $k = 1$ to $\lceil r/j \rceil$) **do**
7. $s := \lfloor (k - 1)j \rfloor$;
8. $mergeImpr(A, s + 1, s + \lceil j/2 \rceil, s + j)$;
9. $j := 2j$;



Original Sequence
18 26 32 6 43 15 9 1
Sorted Sequence
1 6 9 15 18 26 32 43
18 26 32 6 43 15 9 1
6 18 26 32
1 9 15 43
18 26 32 6 43 15 9 1
18 26
6 32
15 43
1 9
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
$\lceil \log_2 n \rceil$ phases

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:

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

$\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS, Chapter 4)

Recurrences – I

Recurrence Relations

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- **Solution Methods** (Chapter 4)
  - Substitution Method.
  - Recursion-tree Method.
  - Master Method.
- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.
  - **Ex:** Divide and Conquer.

$$T(n) = \Theta(1) \qquad \text{if } n \le c$$
$$T(n) = a\,T(n/b) + D(n) \qquad \text{otherwise}$$

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.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

Example – Exact Function

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:** $T(n) = 2\,T(n/2) + n$

$$= 2\,((n/2)\lg(n/2) + (n/2)) + n$$
$$= n\,(\lg(n/2)) + 2n$$
$$= n \lg n - n + 2n$$
$$= n \lg n + n$$

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

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

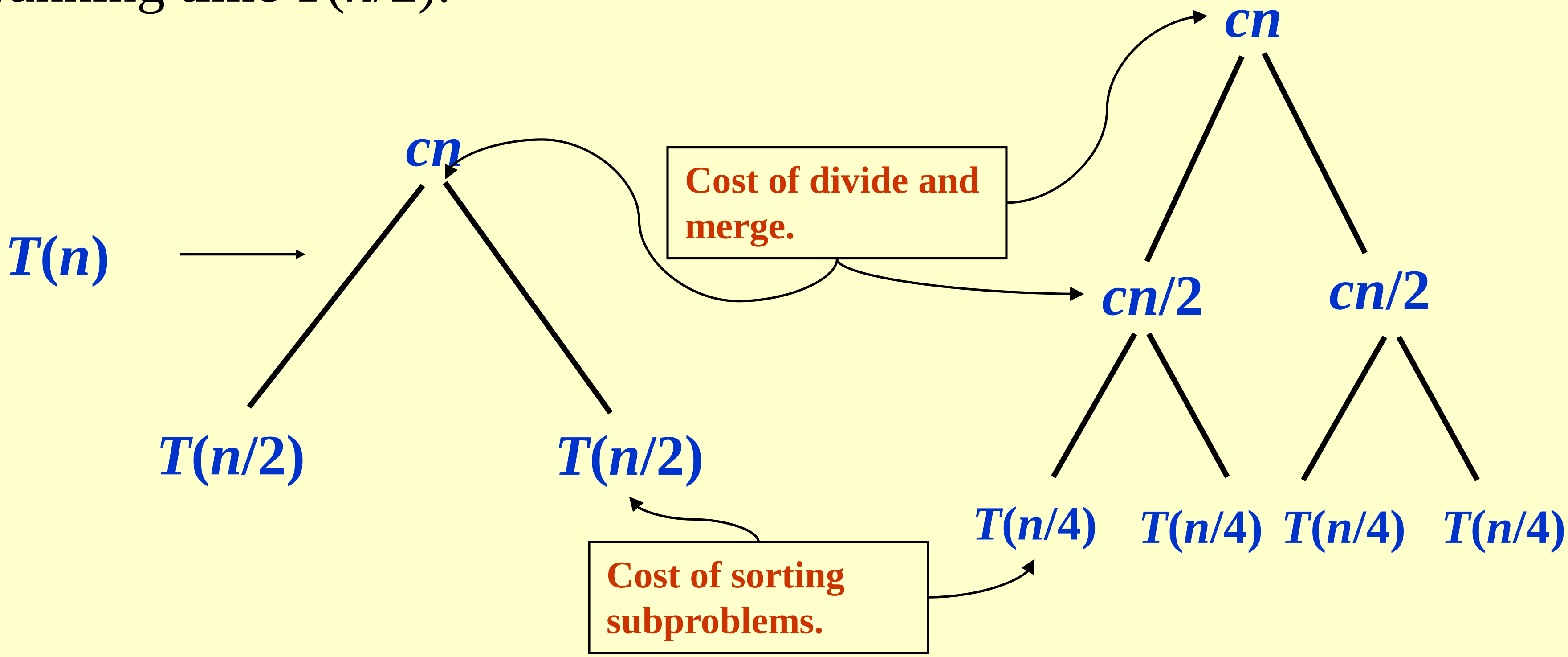
$$T(n) = 2T(n/2) + \boldsymbol{cn} \qquad \text{if } n > 1$$

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

Recursion Tree for Merge Sort

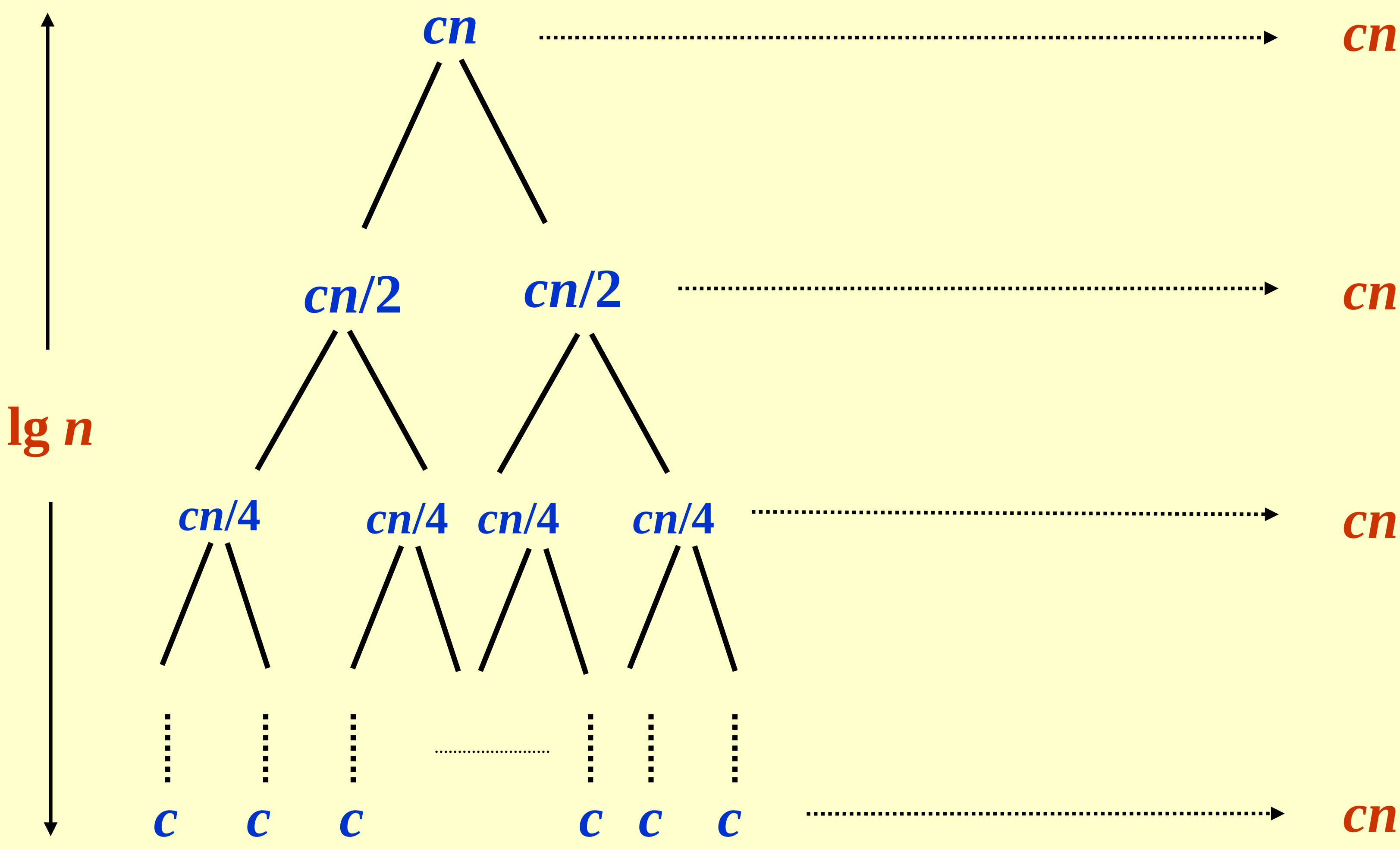
For the original problem, we have a cost of *cn*, plus two subproblems each of size (n/2) and running time $T(n/2)$.

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

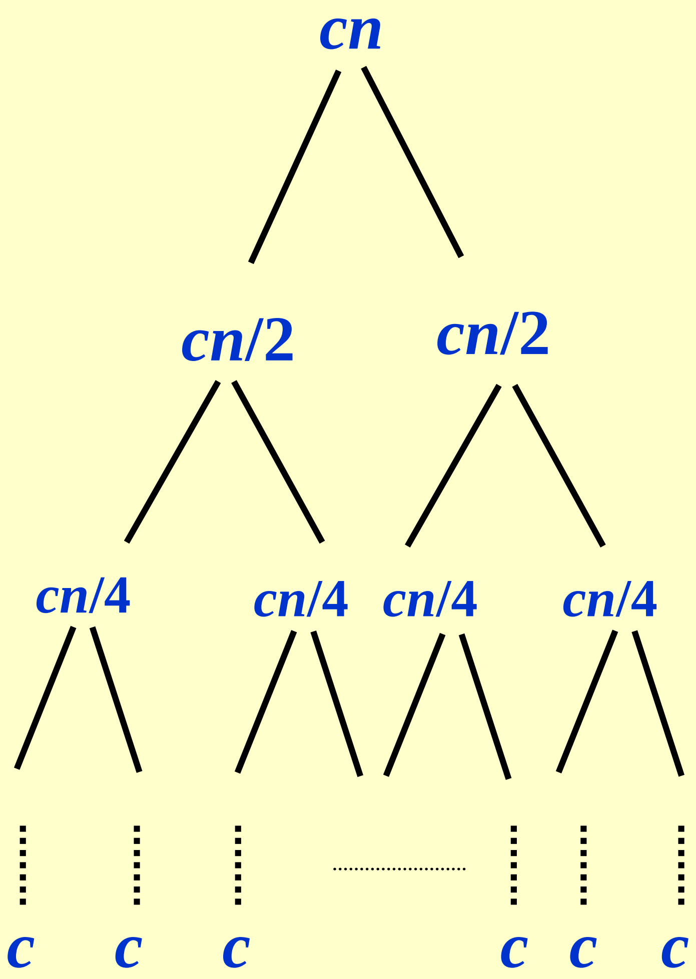
Continue expanding until the problem size reduces to 1.



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

Other Examples

- Use the recursion-tree method to determine a guess for the recurrences
  - $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.
  - $T(n) = T(n/3) + T(2n/3) + O(n)$.

Recursion Trees – Caution Note

- Recursion trees only generate guesses.
  - Verify guesses using substitution method.
- A small amount of "sloppiness" can be tolerated. Why?
- If careful when drawing out a recursion tree and summing the costs, it can be used as direct proof.

The Master Method

- Based on the Master theorem.
- "Cookbook" approach for solving recurrences of the form

  $$T(n) = aT(n/b) + f(n)$$

  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, but we ignore floors and ceilings. Why?
- Requires memorization of three cases.

The Master Theorem

Theorem 4.1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. $T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

We'll return to recurrences as we need them…