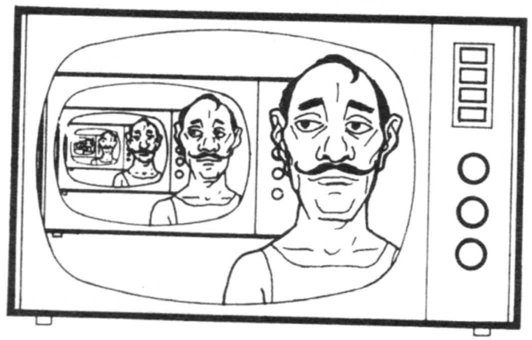
Recursion and the divide and conquer paradigm

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Definition

- **Recursive** procedure
  - Inside its definition there is a call to the procedure itself (*direct recursion*)
  - There is a call to at least one procedure that, directly or indirectly, calls the procedure itself (*indirect recursion)*
- **Recursive** algorithm
  - Based on recursive procedures

Definition



- The solution to a problem S applied to data D is recursive if we can express it as
  - $S(D) = f(S(D'))$ iff $D \neq D_0$
  - $S(D_0) = S_0$ otherwise

Termination condition

D’ simpler than D

Rationale

- Nature of many problems
  - Solution of sub-problems similar to the initial one, though simpler and smaller
  - Combination of partial solutions to obtain the solution of the initial problem
  - Recursion as the basis for the di problem-solving paradigm known as **divide and conquer**
- Mathematical elegance of the solution

Termination condition

All algorithms must eventually terminate $\Rightarrow$ finite recursion

Simple and solvable sub-problems

- Trivial
  - E.g., sets with just 1 element
- Valid choices exhausted
  - E.g., the adjacency list in the graph is over

The Divide and Conquer Paradigm

- Divide
  - Starting from a problem of size n generate $a \geq 1$ **independent** problems of size $n' < n$
- Conquer
  - Solve an elementary problem (termination condition)
- Combine
  - Build a global solution combining partial solutions

The Divide and Conquer Paradigm

Solve (Problem)

- If problem elementary
  - Solution = Trivial_solve (Problem)
- else
  - $\text{Subproblem}_{1,2,3,\ldots,a}$ = Divide (Problem)
  - For each Subproblem_i
    - Subsolution_i = Solve (Subproblem_i)
  - Return Solution = Combine ($\text{Subsolution}_{1,2,3,\ldots,a}$)

Termination condition

Recursive call

"a" subproblems, each smaller than the original one

The Divide and Conquer Paradigm

- Values of a
  - a=1: linear recursion
  - a>1: multi-way recursion

b = n / n′

- Values of n′: at each step problem size is reduced by
  - A constant value, not always the same for all subproblems
  - A constant factor, in general the same for all subproblems
  - A variable quantity, often difficult to estimate

The Divide and Conquer Paradigm

Terminology found in the literature

- Divide and conquer
  - a>1 and in general constant reduction factor or value
- Decrease and conquer
  - a=1 and in general constant reduction value

A First Example: Array Split

- Specifications
  - Given an array of $n=2^k$ integers
  - Recursively partition it in sub-arrays half the size, until the termination condition is reached (1 cell sub-array)
  - Print-out all generated partitions on standard output

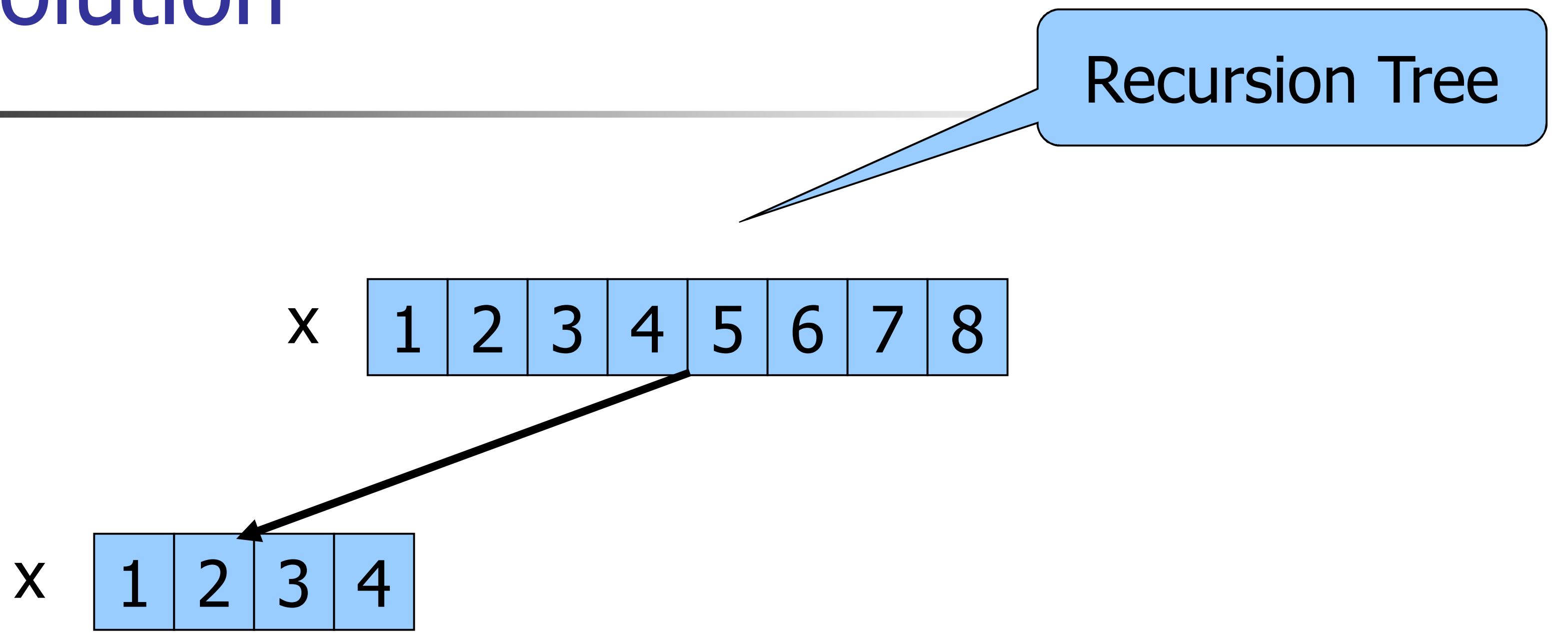
Divide and conquer
$a = 2$ and $b = 2$

Solution

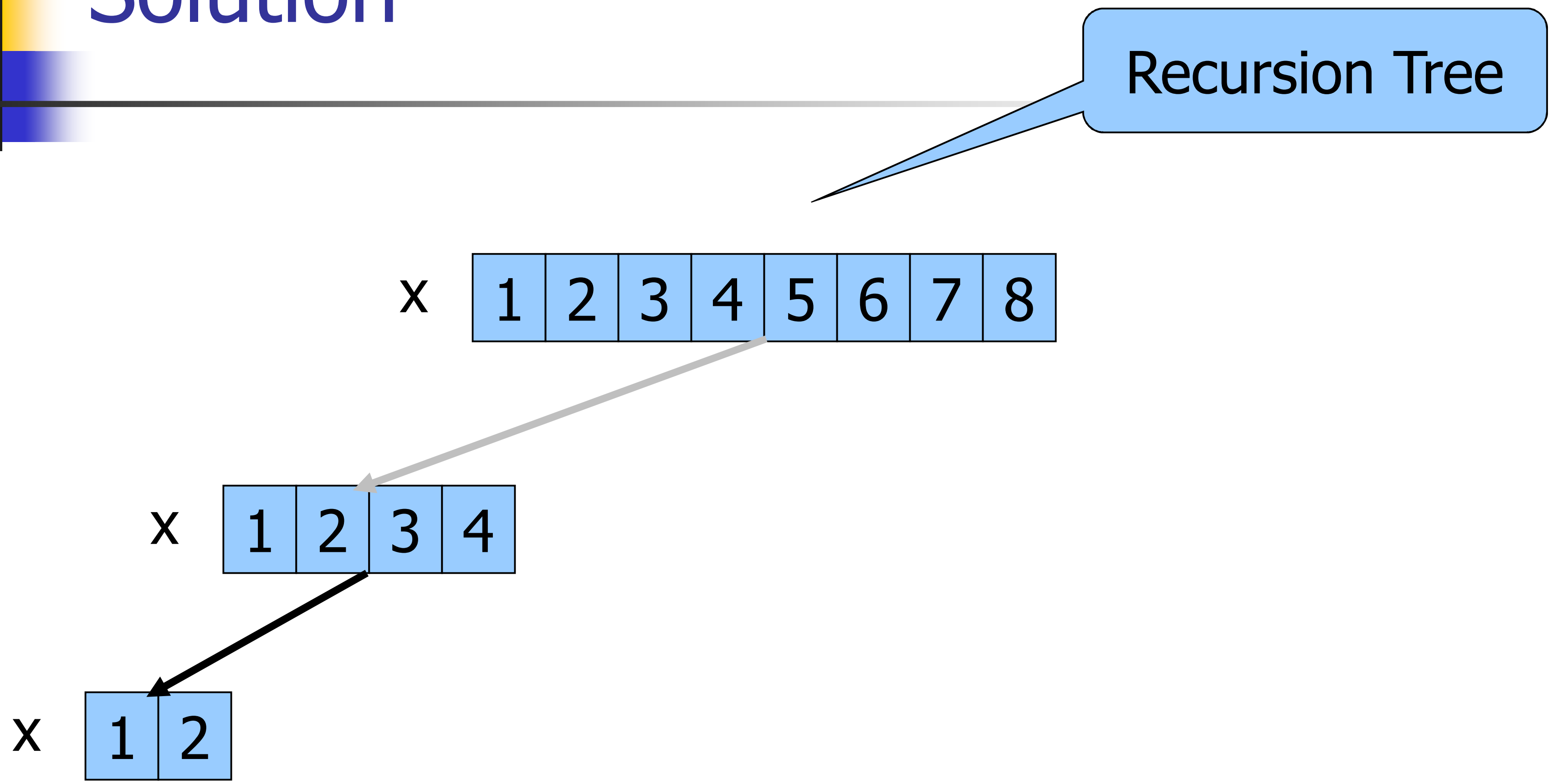
Recursion Tree

x	1	2	3	4	5	6	7	8

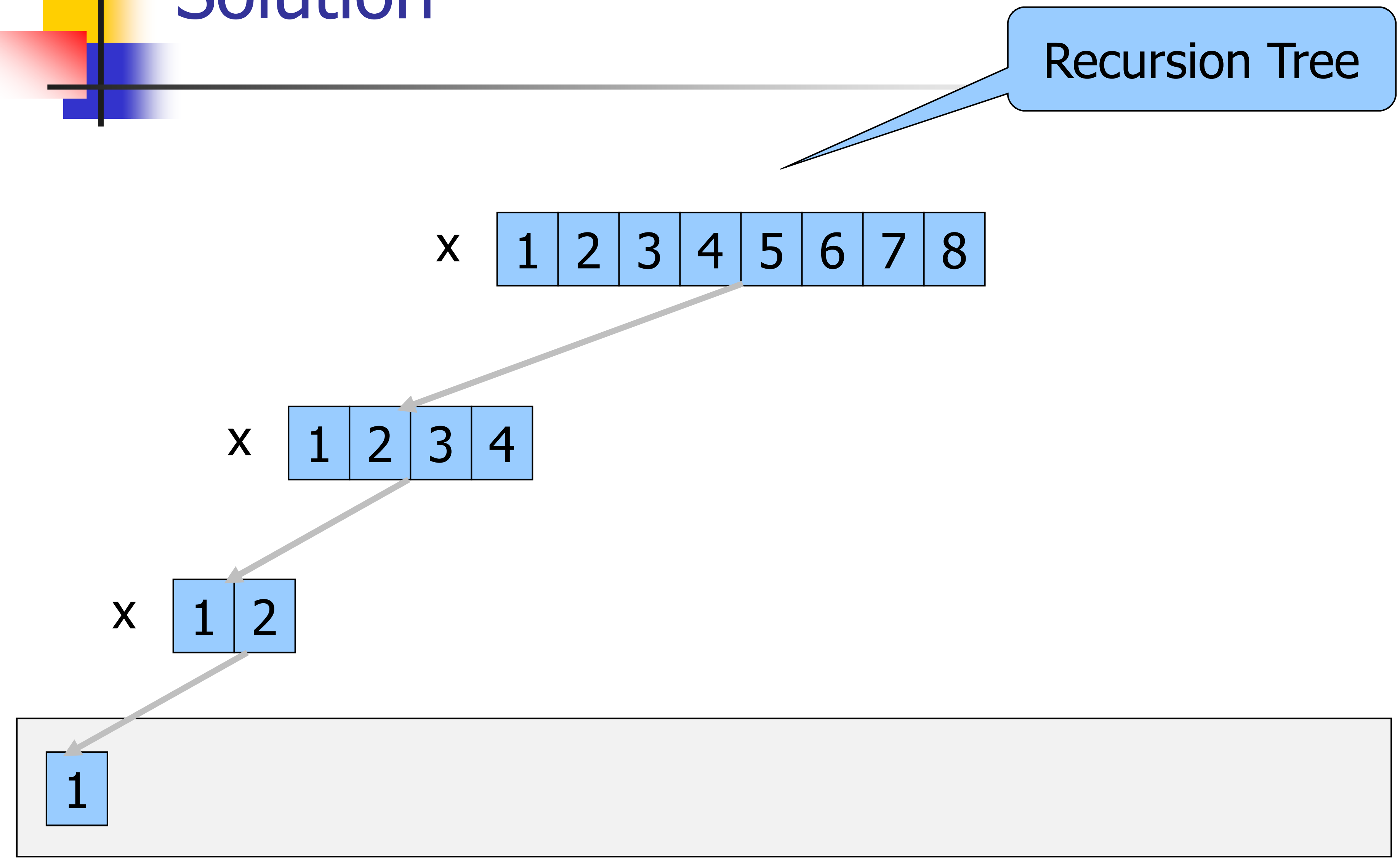
Solution



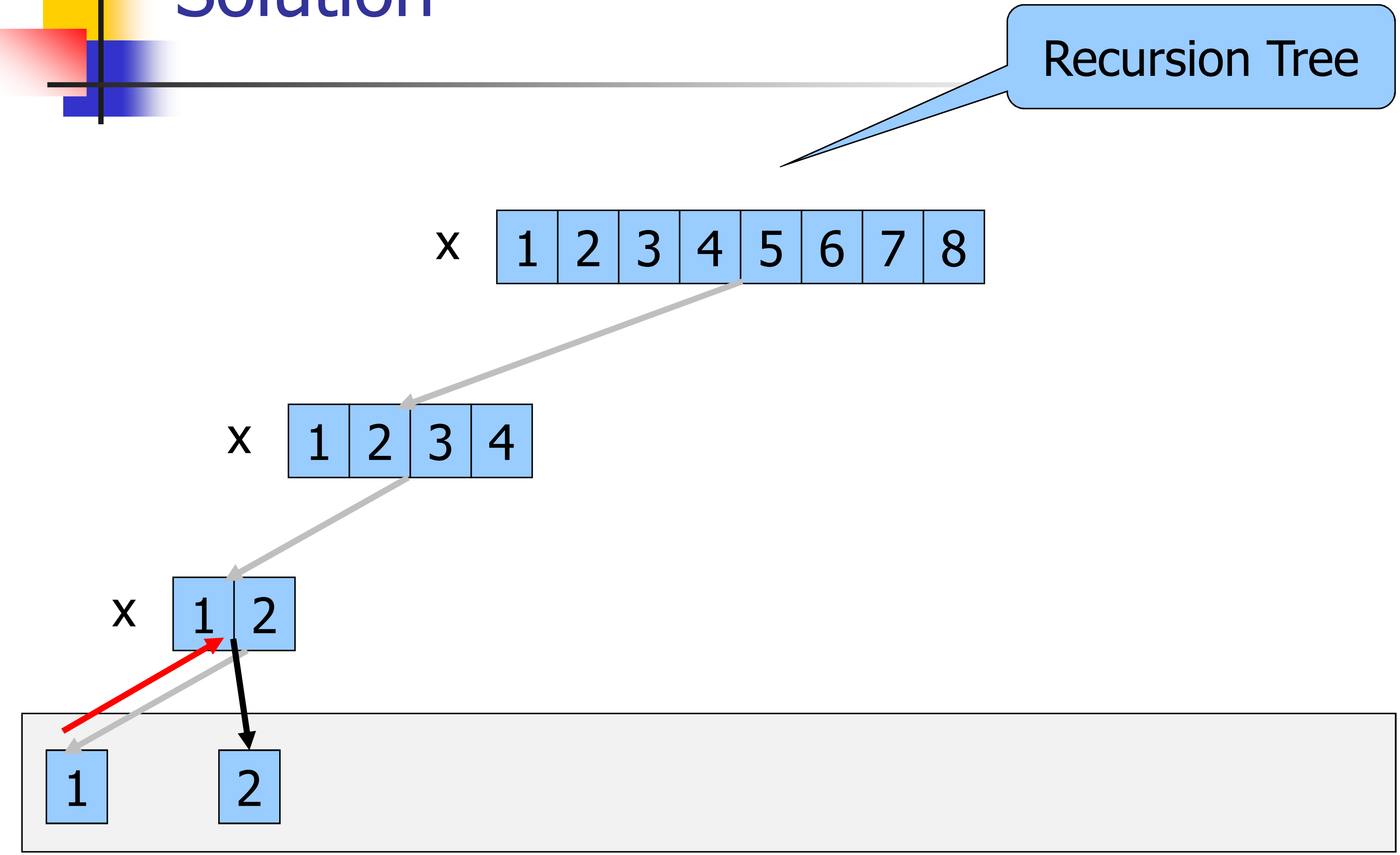
Solution



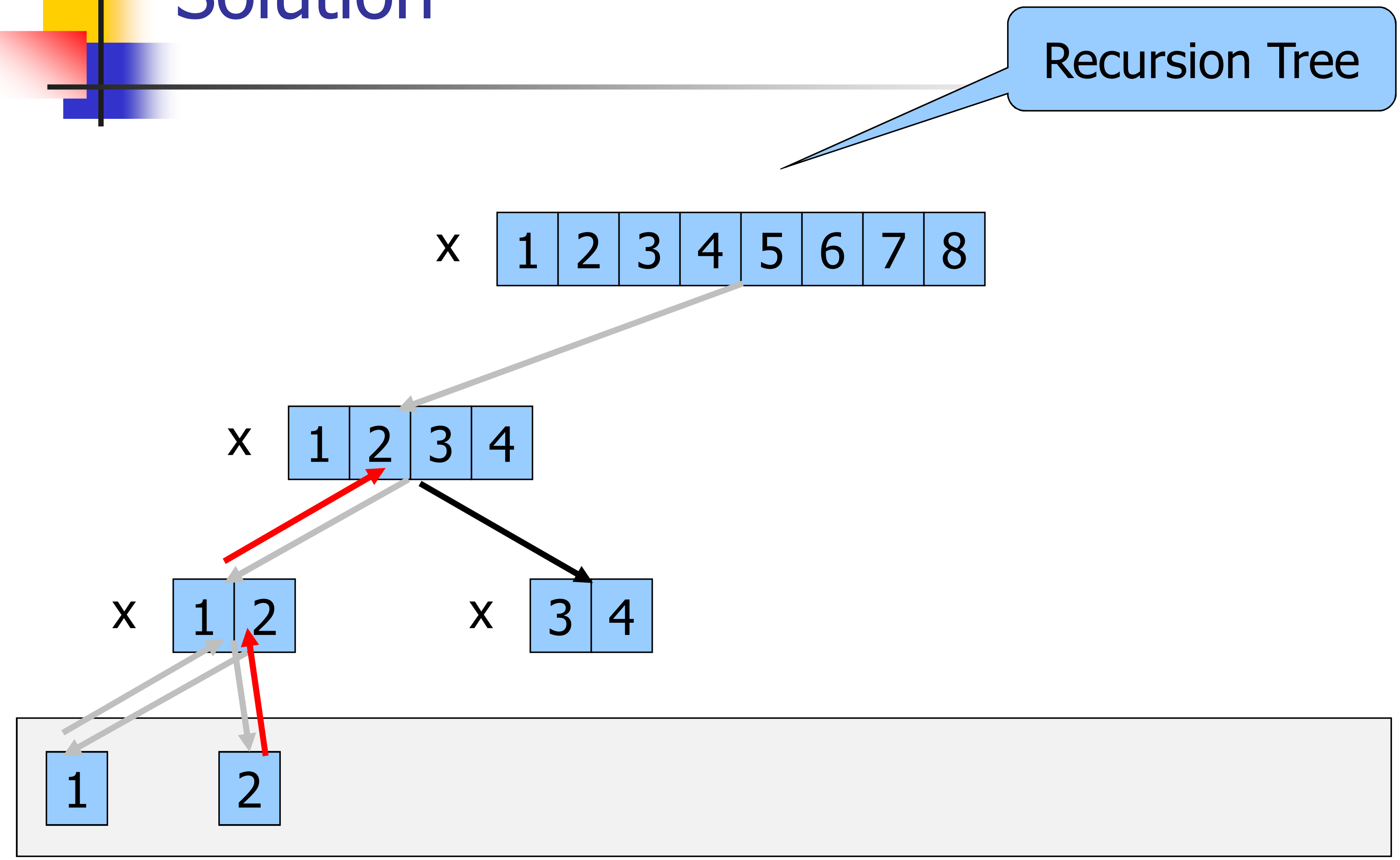
Solution



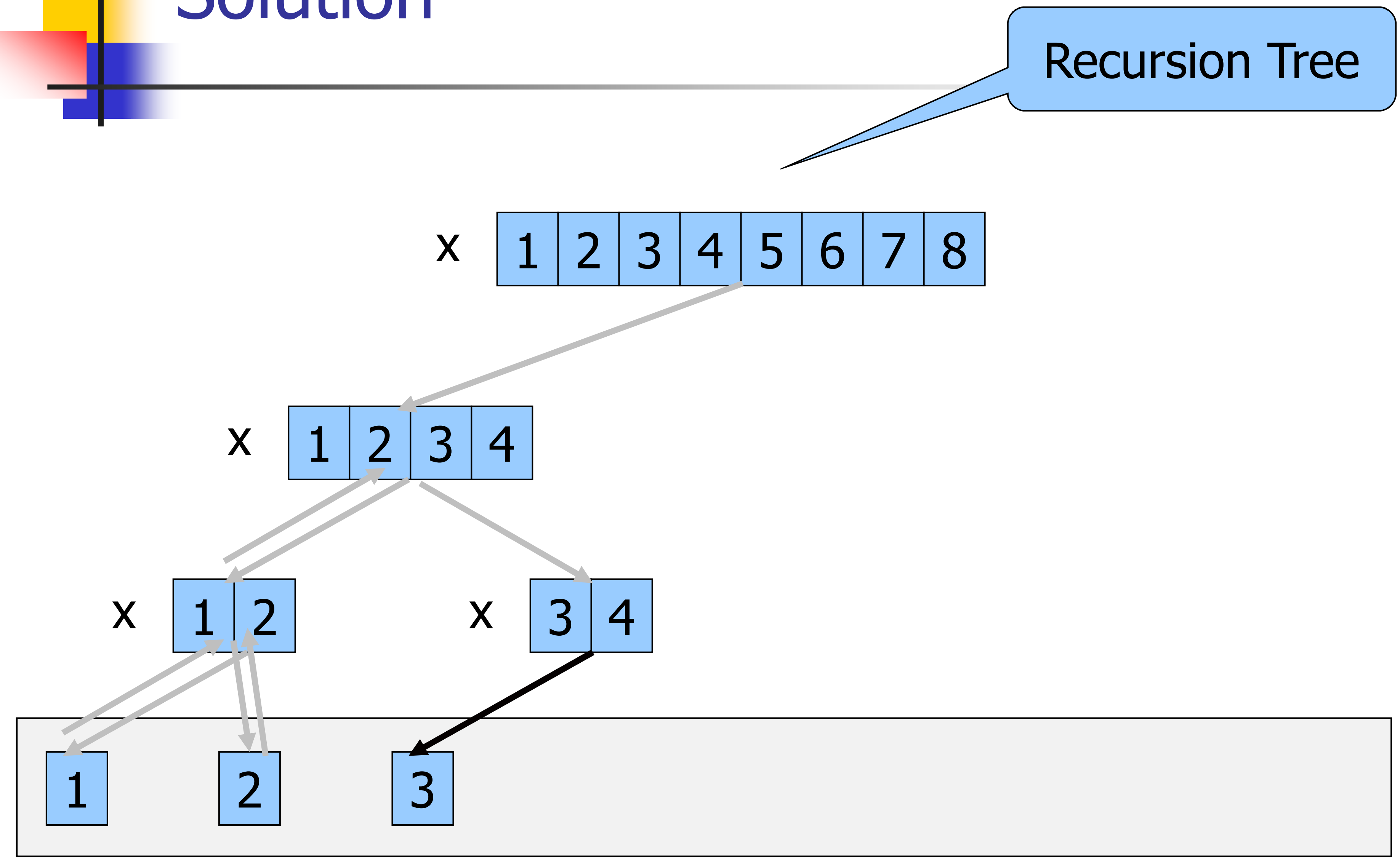
Solution



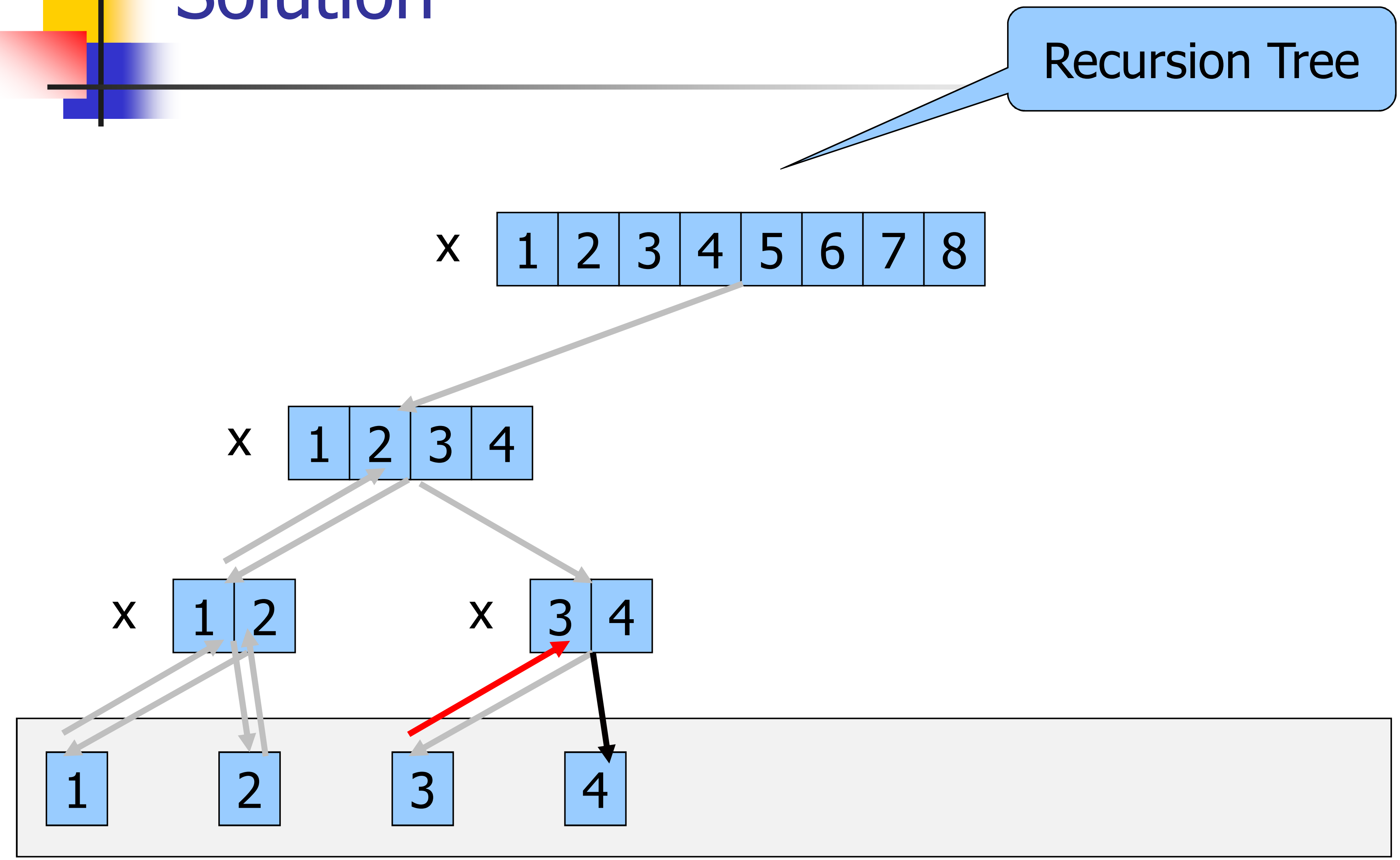
Solution



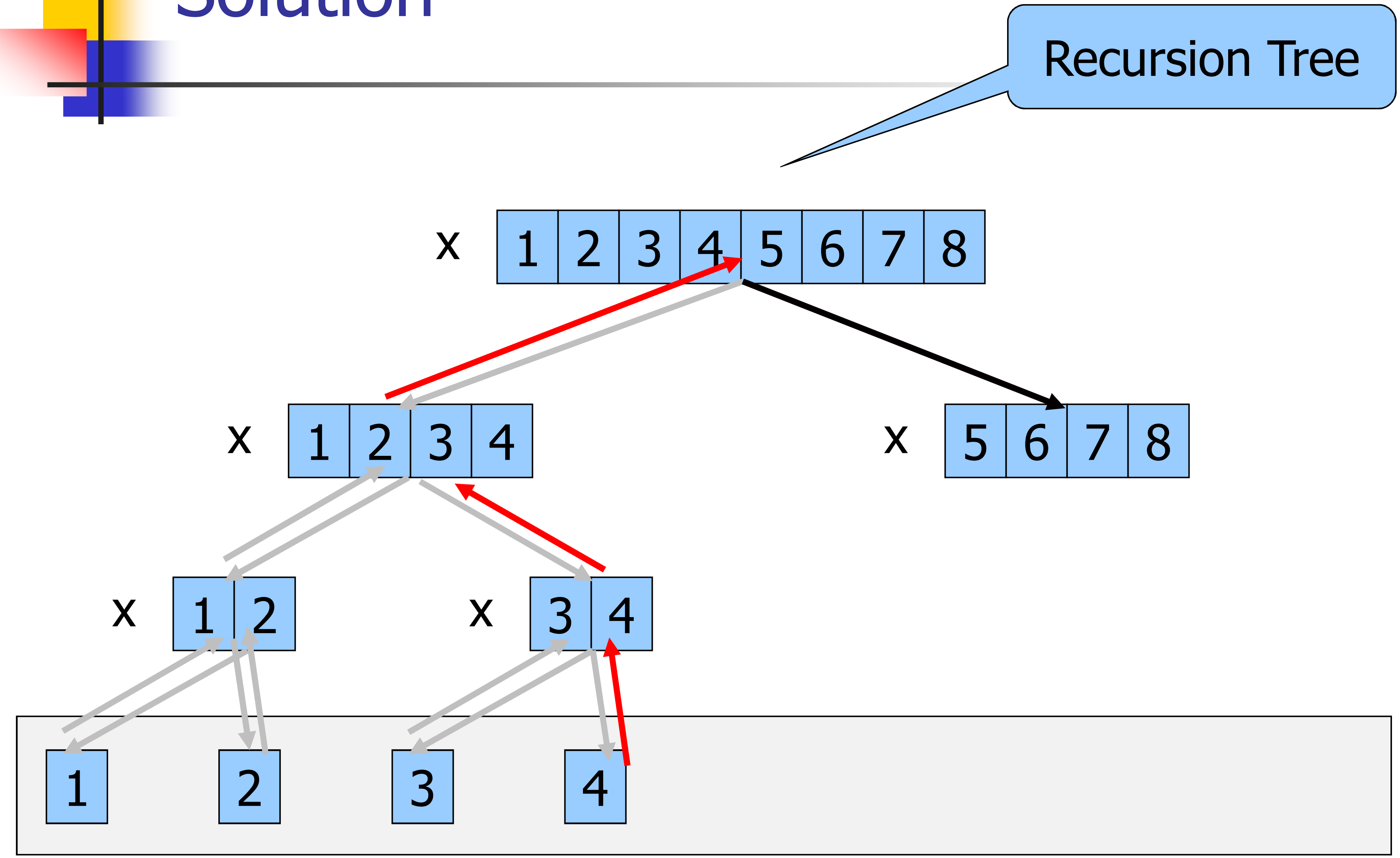
Solution



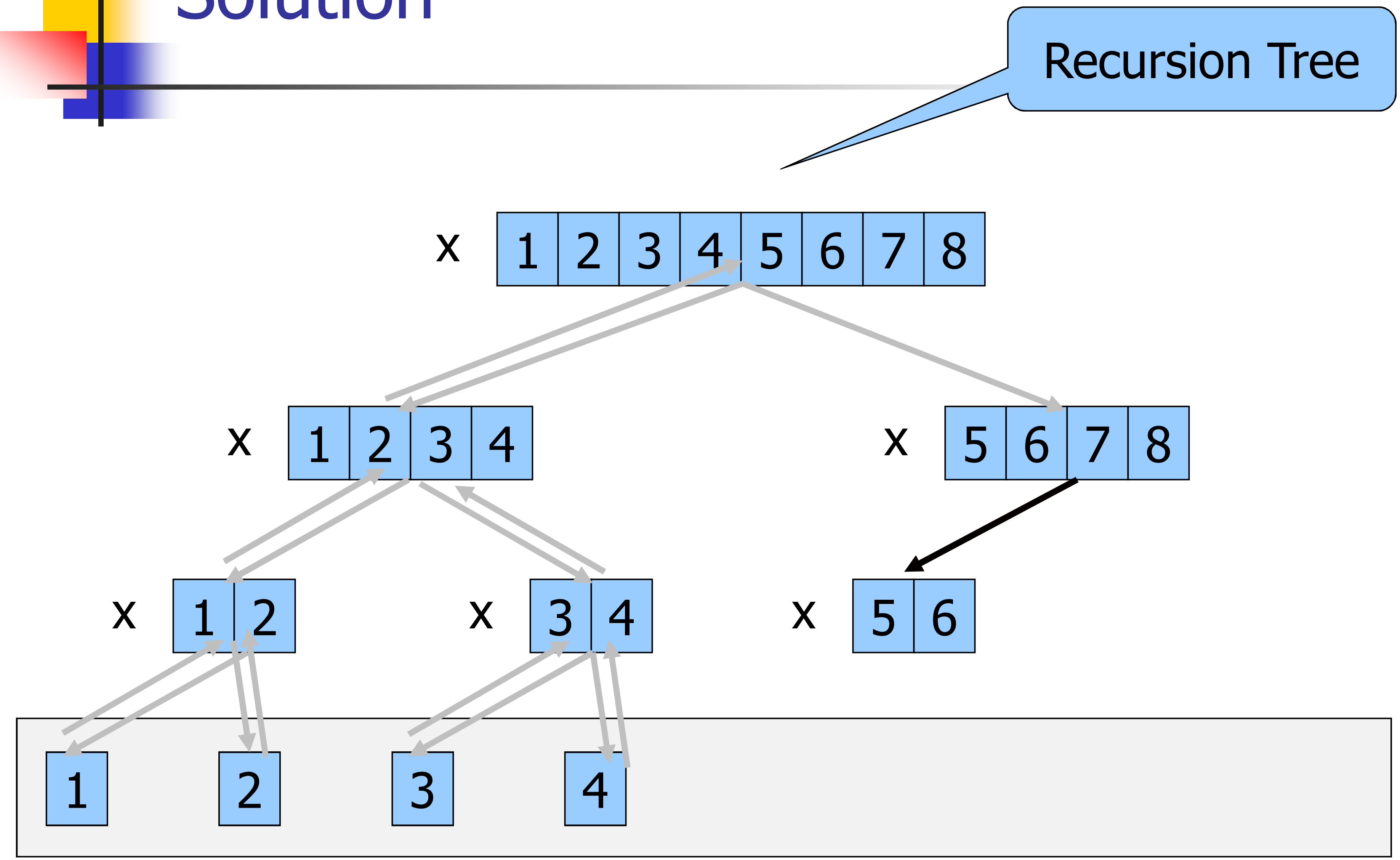
Solution



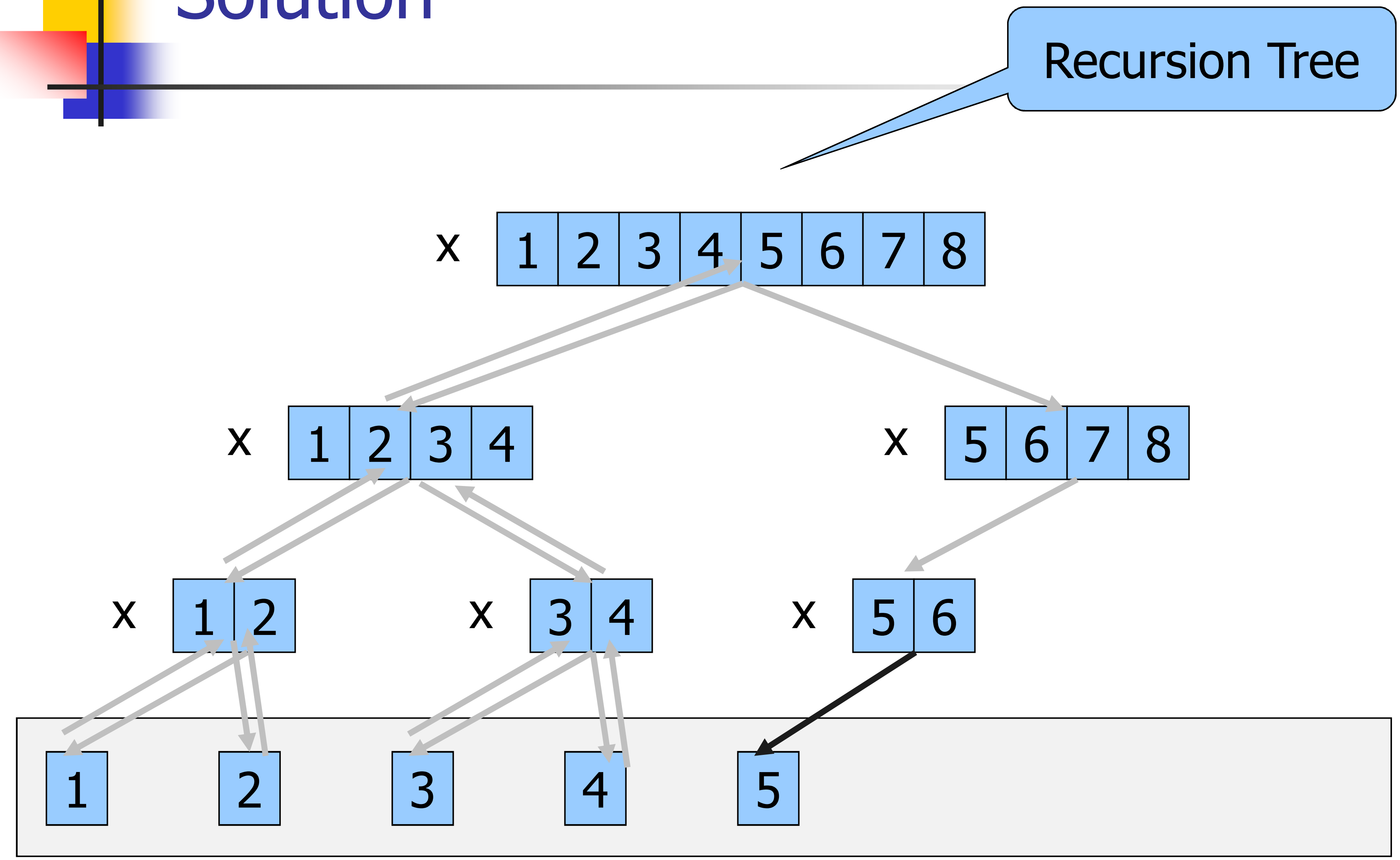
Solution



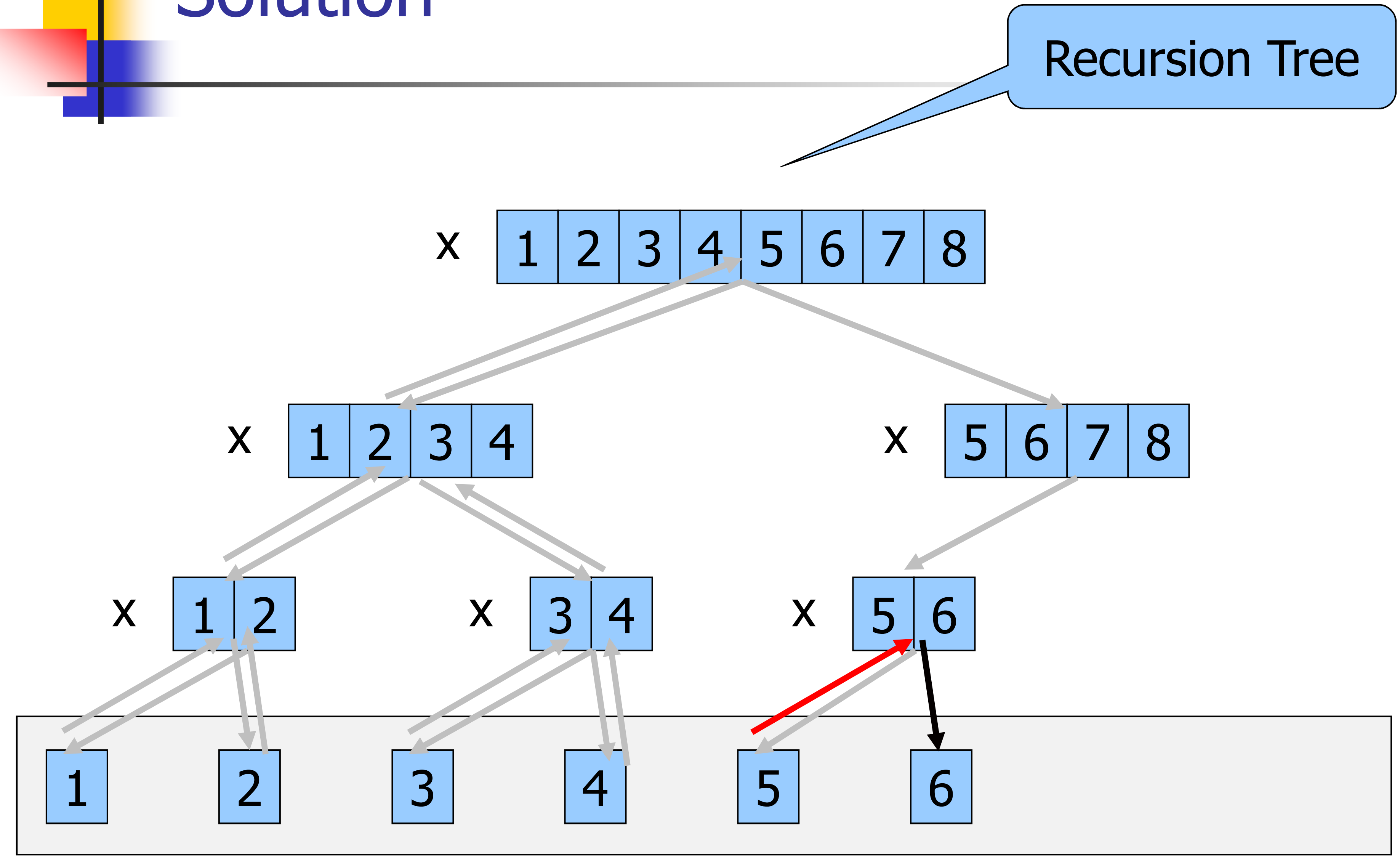
Solution



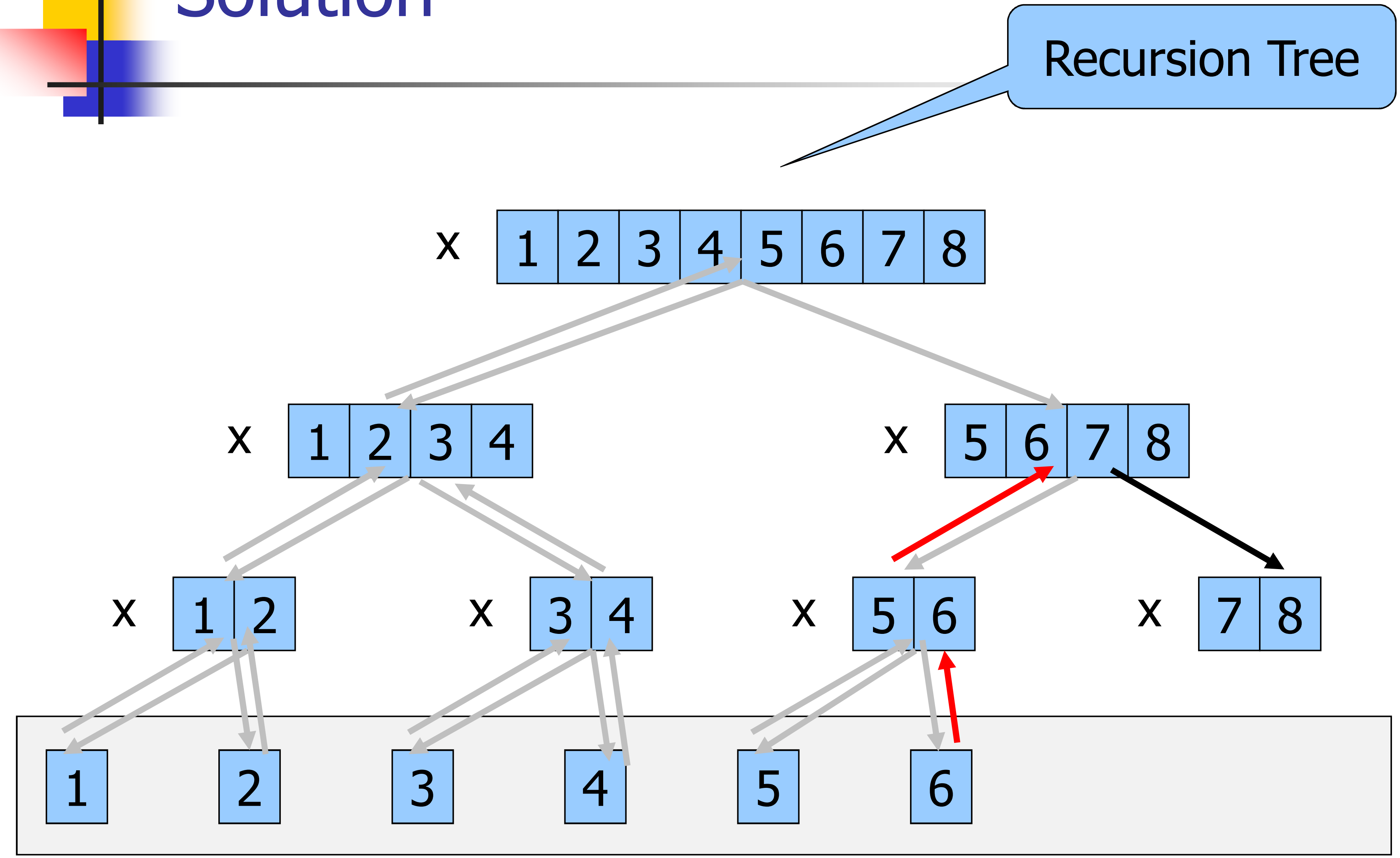
Solution



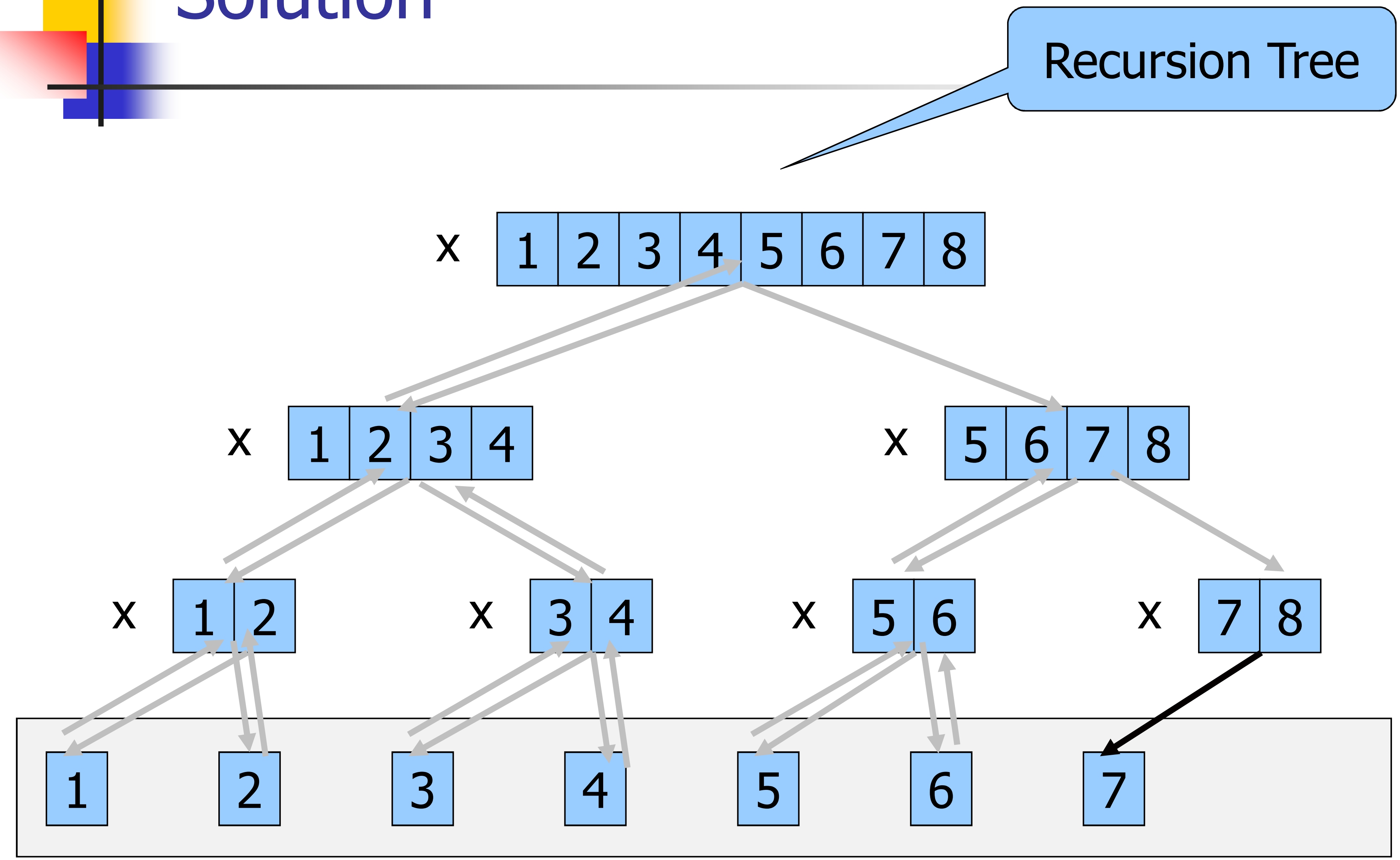
Solution



Solution

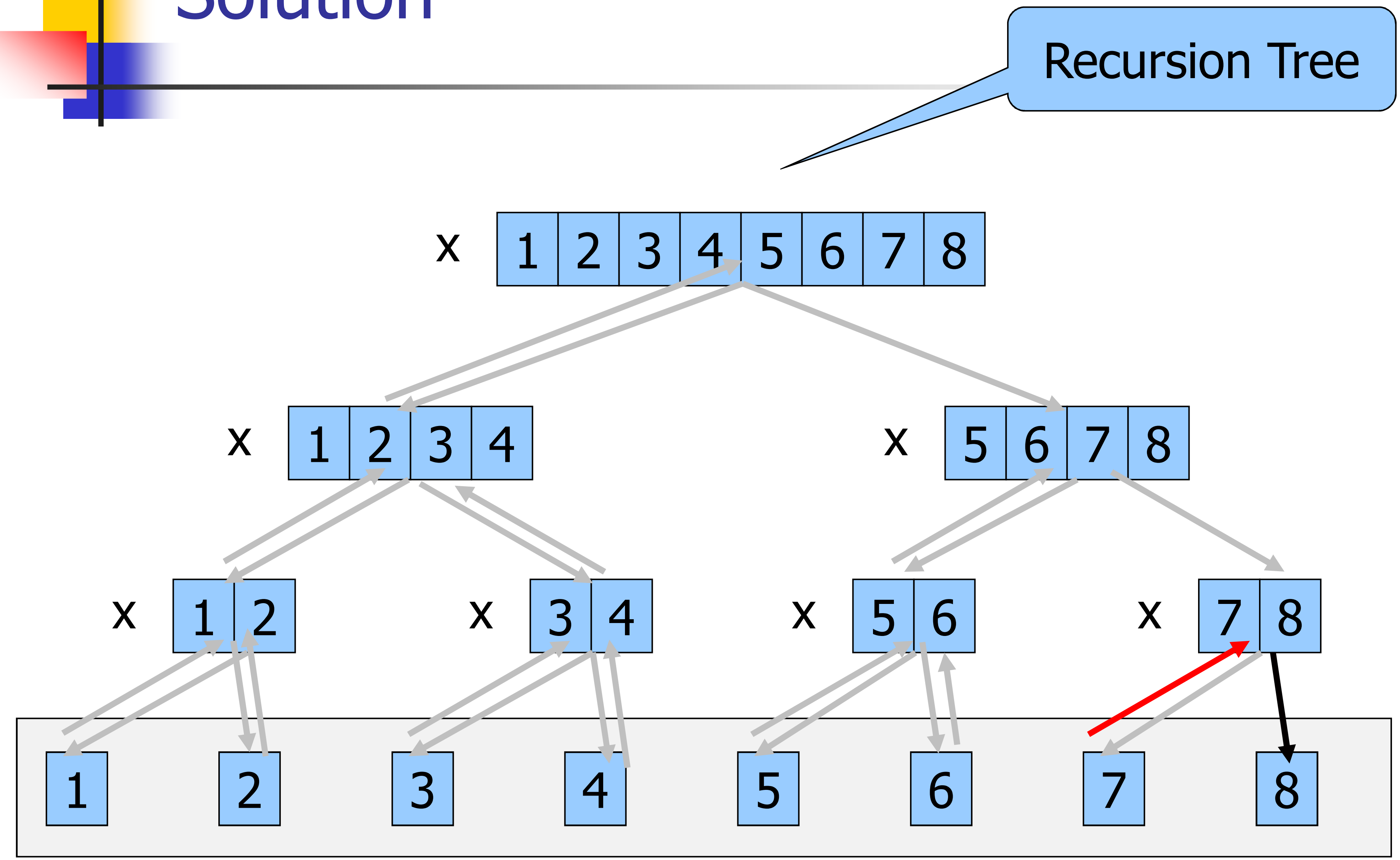

Recursion Tree
x 1 2 3 4 5 6 7 8
x 1 2 3 4
x 5 6 7 8
x 1 2
x 3 4
x 5 6
x 7 8
1
2
3
4
5
6

Solution



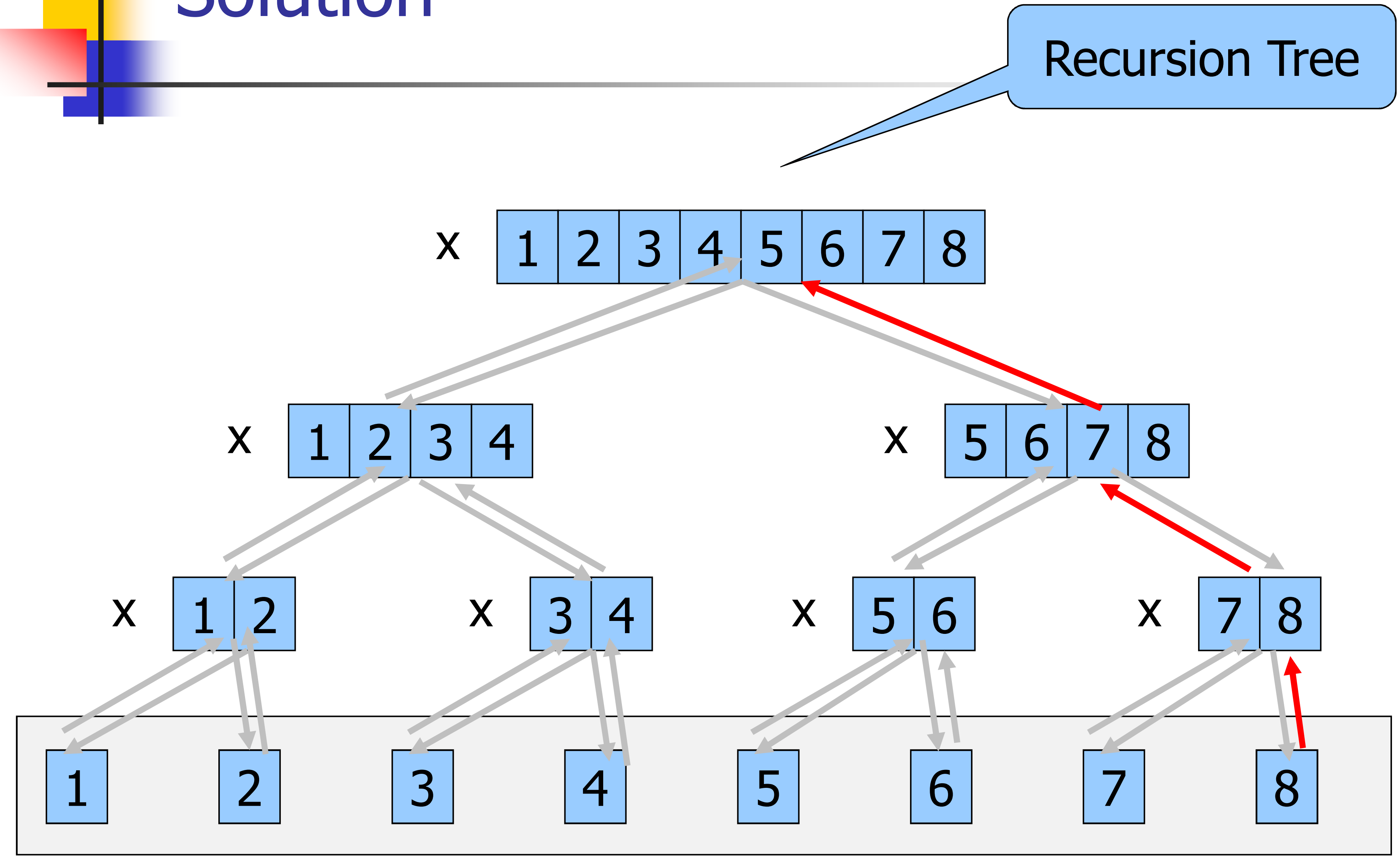
Recursion Tree
x 1 2 3 4 5 6 7 8
x 1 2 3 4
x 5 6 7 8
x 1 2
x 3 4
x 5 6
x 7 8
1
2
3
4
5
6
7

Solution



Recursion Tree
x 1 2 3 4 5 6 7 8
x 1 2 3 4
x 5 6 7 8
x 1 2
x 3 4
x 5 6
x 7 8
1
2
3
4
5
6
7
8

Solution



Termination elements (not printed)

Solution

```
void show(int x[], int l, int r) {
  int i, c;

  if (l >= r) {
    return;
  }

  printf("x = ");
  for (i=l; i<=r; i++)
    printf("%d", x[i]);
  printf("\n");

  c = (r+l)/2;

  show(x, l, c);
  show(x, c+1, r);

  return;
}
```

Termination condition

Array print

Recursions (left and right)

Complexity Analysis

A **recursion equation** expresses T(n) in terms of

- D(n)
  - Cost of dividing
- Cost of execution time for smaller inputs (recursion)
- C(n)
  - Cost of recombination

We assume unit cost ($\Theta(1)$) for solving the elementary problem

Complexity Analysis

- If
  - a is the number of subproblems originating from the "divide" phase
  - b is the reduction factor, thus n/b is the size of each subproblem
- The recurrence equation has the following form

$$T(n) = \overbrace{D(n)}^{\text{divide}} + \overbrace{a\,T(n/b)}^{\text{recur}} + \overbrace{C(n)}^{\text{combine}} \qquad n > c$$

$$T(n) = \Theta(1) \qquad n \leq c$$

Complexity Analysis

T(n)

Solve(Problem)

- If problem elementary
  - Solution = Trivial_solve(Problema)
- else
  - $\text{Subproblem}_{1,2,3,\dots,a}$ = Divide (Problem)
  - For each Subproblem_i:
    - Subsolution_i = Solve (Subproblem_i)
  - Return Solution = Combine ($\text{Subsolution}_{1,2,3,\dots,a}$)

$\Theta(1)$

a subproblems

D(n)

T(n/b)

C(n)

Complexity Analysis

If

- a is the number of subproblems originating from the divide phase
- Reduction amounts to k_i, an amount that may very at each step

the recurrence equation has the following form

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-ki) + C(n) \qquad n > c$$

$$T(n) = \Theta(1) \qquad n \leq c$$

Complexity Analysis

T(n)

Solve(Problem)

- If problem elementary
  - Solution = Trivial_solve(Problema)

$\Theta(1)$

D(n)

- else
  - $\text{Subproblem}_{1,2,3,\ldots,a}$ = Divide (Problem)
  - For each Subproblem_i:
    - Subsolution_i = Solve (Subproblem_i)
  - Return Solution = Combine ($\text{Subsolution}_{1,2,3,\ldots,a}$)

a subproblems

C(n)

$T(n-k_i)$

Array Split: Complexity Analysis

divide and conquer
$a = 2$ $b = 2$

- Complexity analysis
  - $D(n) = \Theta(1)$
  - $C(n) = \Theta(1)$
  - $a = 2, b = 2$
- Recurrence equation

Divide – Recur - Combine

  - $T(n) = D(n) + a\, T(n/b) + C(n)$
- That is
  - $T(n) = 2T(n/2) + 1$ $\quad n > 1$
  - $T(1) = 1$ $\quad n=1$

Array Split: Complexity Analysis

- Resolution by unfolding
  - $T(n) = 1 + 2T(n/2)$
  - $T(n/2) = 1 + 2T(n/4)$
  - $T(n/4) = 1 + 2T(n/8)$
  - …

Termination condition
$n/2^i = 1$
$i = \log_2 n$

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

- Replacing in T(n)
  - $T(n) = 1 + 2 + 4 + 2^3 T(n/8)$

$$= \sum_{i=0}^{\log_2 n} 2^i = (2^{\log_2 n + 1} - 1)/(2-1)$$

$$= 2 \cdot 2^{\log_2 n} - 1 = 2n-1$$

- Thus
  - $T(n) = O(n)$