COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

The Activity Selection Problem

You are given several activities with start and finish times. You can perform only one activity at a time. The goal is to select the maximum number of non-overlapping activities.

Activity	Start	Finish
A1	1	3
A2	4	6
A3	2	5
A4	6	7
A5	5	8
A6	8	9

The Activity Selection Problem

You are given several activities with start and finish times. You can perform only one activity at a time. The goal is to select the maximum number of non-overlapping activities.

Activity	Start	Finish
A1	1	3
A2	4	6
A3	2	5
A4	6	7
A5	5	8
A6	8	9

Each activity i has a start time s[i] and a finish time f[i].
Two activities i and j can be scheduled if their time intervals do not overlap, i.e. f[i] ≤ s[j] or f[j] ≤ s[i].

The Activity Selection Problem

Idea 2

The Activity Selection Problem

Idea 2

1. Sort all activities by their finish time
2. Select the first activity (the one that finishes earliest)
3. For each subsequent activity:
    - If its start time is greater than or equal to the finish time of the last selected activity, select it
4. Continue until all activities are checked

The Activity Selection Problem

Idea 2

1. Sort all activities by their finish time
2. Select the first activity (the one that finishes earliest)
3. For each subsequent activity:
    - If its start time is greater than or equal to the finish time of the last selected activity, select it
4. Continue until all activities are checked

Works ?

Time ?
— Once sorted, you need to go through all activities for selection.
Total time = Sorting + Selection = Sorting + n.

The Activity Selection Problem

Activity	Start	Finish
A1	1	3
A2	4	6
A3	2	5
A4	6	7
A5	5	8
A6	8	9

Sorted as per finish time →

Activity	Start	Finish
A1	1	3
A3	2	5
A2	4	6
A4	6	7
A5	5	8
A6	8	9

After sorting by finish time: A1, A2, A4, A5, A6

Select A1 → (finish = 3)

A2 starts at 4 ≥ 3 → select A3 (finish = 6)

A4 starts at 6 ≥ 6 → select A4 (finish = 7)

A6 starts at 8 ≥ 7 → select A6 (finish = 9)

The Activity Selection Problem

1. Sort all activities by their finish time
2. Select the first activity (the one that finishes earliest)
3. For each subsequent activity:
    - If its start time is greater than or equal to the finish time of the last selected activity, select it
4. Continue until all activities are checked

Greedy approach

the idea relies on the principle of making the locally optimal choice — always pick the next activity that finishes earliest among the remaining compatible ones.

The Activity Selection Problem

1. Sort all activities by their finish time
2. Select the first activity (the one that finishes earliest)
3. For each subsequent activity:
    - If its start time is greater than or equal to the finish time of the last selected activity, select it
4. Continue until all activities are checked

Greedy approach

the idea relies on the principle of making the locally optimal choice — always pick the next activity that finishes earliest among the remaining compatible ones.

always choosing the next piece that looks best at the moment — without worrying about the consequences for future choices.

Greedy Approach

Greedy algorithms make **locally optimal** choices at each step with the **hope** that these lead to a **globally optimal solution**.

Local Optimum: The best solution within a small or limited part of the problem. What looks best right now (within a small neighborhood)

Global Optimum: The best solution among all possible solutions of the entire problem. Best overall, across the entire search space.

Recall!

Given a set of n activities $A = \{1,2,\ldots,n\}$ where each activity i has a start time s_i and a finish time f_i, with $s_i < f_i$. Two activities i and j are compatible if their time intervals do not overlap, i.e. $[s_i,f_i) \cap [s_j,f_j) = \emptyset$.

Recall!

Given a set of n activities $A = \{1,2,\ldots,n\}$ where each activity i has a start time s_i and a finish time f_i, with $s_i < f_i$. Two activities i and j are compatible if their time intervals do not overlap, i.e. $[s_i,f_i) \cap [s_j,f_j) = \emptyset$.

Let: $G = \{g_1, \ldots, g_k\}$: activities chosen by the greedy algorithm.

$O = \{o_1, \ldots, o_m\}$: activities in an optimal solution (maximum number possible).

We must show $k = m$. That is, the greedy algorithm picks as many activities as any optimal method.

Recall!

Given a set of n activities $A = \{1,2,\ldots,n\}$ where each activity i has a start time s_i and a finish time f_i, with $s_i < f_i$. Two activities i and j are compatible if their time intervals do not overlap, i.e. $[s_i,f_i) \cap [s_j,f_j) = \emptyset$.

Let: $G = \{g_1, \ldots, g_k\}$: activities chosen by the greedy algorithm.

$O = \{o_1, \ldots, o_m\}$: activities in an optimal solution (maximum number possible).

We must show $k = m$. That is, the greedy algorithm picks as many activities as any optimal method.

The greedy algorithm picks the activity g_1 that finishes earliest.

The optimal solution may have picked some activity $o_1$that finishes later or at the same time.

$f(g_1) <= f(o_1)$ — Every activity compatible with o_1 is also compatible with g_1

Recall!

Given a set of n activities $A = \{1,2,\ldots,n\}$ where each activity i has a start time s_i and a finish time f_i, with $s_i < f_i$. Two activities i and j are compatible if their time intervals do not overlap, i.e. $[s_i,f_i) \cap [s_j,f_j) = \emptyset$.

Let: $G = \{g_1, \ldots, g_k\}$: activities chosen by the greedy algorithm.

$O = \{o_1, \ldots, o_m\}$: activities in an optimal solution (maximum number possible).

We must show $k = m$. That is, the greedy algorithm picks as many activities as any optimal method.

The greedy algorithm picks the activity g_1 that finishes earliest.

The optimal solution may have picked some activity $o_1$that finishes later or at the same time.

$f(g_1) <= f(o_1)$ — Every activity compatible with o_1 is also compatible with g_1

So $in\ O$, we can replace o_1 by g_1 — same argument applies recursively to $\forall i$

Recall!

Given a set of n activities $A = \{1,2,\ldots,n\}$ where each activity i has a start time s_i and a finish time f_i, with $s_i < f_i$. Two activities i and j are compatible if their time intervals do not overlap, i.e. $[s_i,f_i) \cap [s_j,f_j) = \emptyset$.

Let: $G = \{g_1, \ldots, g_k\}$: activities chosen by the greedy algorithm.
$O = \{o_1, \ldots, o_m\}$: activities in an optimal solution (maximum number possible).
We must show $k = m$. That is, the greedy algorithm picks as many activities as any optimal method.

The greedy algorithm picks the activity g_1 that finishes earliest.
The optimal solution may have picked some activity $o_1$that finishes later or at the same time.
$f(g_1) <= f(o_1)$ — Every activity compatible with o_1 is also compatible with g_1

So $in\ O$, we can replace o_1 by g_1 — same argument applies recursively to $\forall i$

Hence, $k = m$

The Coin Change (Denomination) Problem

Given a set of coin denominations and an amount of money, find the minimum number of coins required to make that amount.

Denominations — {1,5,10,25}
Amount - 30

2 coins (25 + 5)

The Coin Change (Denomination) Problem

Given a set of coin denominations and an amount of money, find the minimum number of coins required to make that amount.

Denominations — {1,5,10,25}
Amount - 30

2 coins (25 + 5)

Formal Specification ?

The Coin Change (Denomination) Problem

Given a set of coin denominations and an amount of money, find the minimum number of coins required to make that amount.

Denominations — {1,5,10,25}
Amount - 30

2 coins (25 + 5)

Given a finite set of coin denominations $D = \{d_1, \ldots, d_n\}$, where each d_i is a positive integer, representing coin value, and target Amount A, integer $A > 0$.

Find a $S = \{c_1, \ldots, c_m\}$ such that $S \subseteq D$, $\Sigma_{i=1}^{m} c_i = A$, m should be as small as possible. If no such S exists then return -1.

The Coin Change (Denomination) Problem

Given a finite set of coin denominations $D = \{d_1, \ldots, d_n\}$, where each d_i is a positive integer, representing coin value, and target Amount A, integer $A > 0$.

Find a $S = \{c_1, \ldots, c_m\}$ such that $S \subseteq D$, $\Sigma_{i=1}^{m} c_i = A$, m should be as small as possible. If no such S exists then return -1.

Approach 1.

1. Generate **all possible subsets** of the d coins.
2. For each subset:
   - Check if total sum = A.
   - If yes, count the coins in it.
3. Keep track of the subset with the smallest count.
4. If none exists, return -1.

} Brute force approach

The Coin Change (Denomination) Problem

Idea — At each step, pick the largest remaining element that does not cause the total to exceed A

1. Sort all coins by their denomination in descending order (largest first).
2. Select the first coin, if its denomination is less than or equal to the target amount A
3. For each subsequent coin:
    - If the sum of the coins already chosen plus the current coin's denomination is less than or equal to A then select the current coin.
4. Continue until all coins have been checked.
5. Output the set of selected coins and their total value.

Greedy Approach!

The Coin Change (Denomination) Problem

$$D = \{8,6,5,4,3\}; A = 11$$

Apply Greedy — S ?

Apply Brute force — S ?

The Coin Change (Denomination) Problem

$$D = \{8,6,5,4,3\}; A = 11$$

Apply Greedy — S ?

Greedy (largest-first): pick 8 → remaining 3 → 3 is available → solution {8,3} (2 coins).

Apply Brute force — S ?

{6,5} or {8,3} (2 coins).

Here — Greedy ties with Optimal

The Coin Change (Denomination) Problem

$$D = \{7,5,4,3\}; A = 9$$

Apply Greedy — S ?

Apply Brute force — S ?

The Coin Change (Denomination) Problem

$$D = \{7,5,4,3\}; A = 9$$

Apply Greedy — S ?

Greedy (largest-first): pick 7 → remaining 2 (no 2 in D) → stops without a solution.

Apply Brute force — S ?

The Coin Change (Denomination) Problem

$$D = \{7,5,4,3\}; A = 9$$

Apply Greedy — S ?

Greedy (largest-first): pick 7 → remaining 2 (no 2 in D) → stops without a solution.

Apply Brute force — S ?

Brute force (or exhaustive subsets): checks all subsets and finds {5,4} = 9 (2 coins).

So a solution exists, but greedy misses it.

The Coin Change (Denomination) Problem

Given a set of coin denominations and an amount of money, find the minimum number of coins required to make that amount.

Given a finite set of coin denominations $D = \{d_1, \ldots, d_k\}$, where each d_i is a positive integer, representing coin value and target Amount A, integer $A > 0$.

and we want to find integers a list $[n_i]$ such that, $n_1 d_1 + n_2 d_2 + \ldots + n_k d_k = A$

$[n_i]$ — can have constraints or may be unconstrainted

The Coin Change (Denomination) Problem

Of course, if we know $[n_i]$ before then the previously discussed algorithm works here

1. Sort all coins **(now each coin c_i will have n_i many copies)** by their denomination in descending order (largest first).
2. Select the first coin, if its denomination is less than or equal to the target amount A
3. For each subsequent coin:
    - If the sum of the coins already chosen plus the current coin's denomination is less than or equal to A then select the current coin.
4. Continue until all coins have been checked.
5. Output the set of selected coins and their total value.

$D = \{1,2,5\}, A = 11$ We have 3 copies of 1, 2 copies of 2 and 5 copies of 5.

$D = \{1,1,1,2,2,5,5,5,5,5\}$

1. Sorted $\{5,5,5,5,5,2,2,1,1,1\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$

2. Sorted$\{5,5,5,5,5,2,2,1,1,1\}$. $-$ $5 <= 6$, pick second 5. Remaining balance $6 - 5 = 1$

3. Sorted $\{5,5,5,5,5,2,2,1,1,1\}$. $-$ $5 <= 6$, pick second 5. Remaining balance $6 - 5 = 1$

4. Sorted$\{5,5,5,5,5,2,2,1,1,1\}$. $-$ $1 <= 1$, pick first 1. Remaining balance $1 - 1 = 0$, Done

$D = \{1,2,5\}, A = 11$ We have 3 copies of 1, 2 copies of 2 and 5 copies of 5.

$D = \{1,1,1,2,2,5,5,5,5,5\}$

1. Sorted $\{5,5,5,5,5,2,2,1,1,1\}$. — $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$

2. Sorted$\{5,5,5,5,5,2,2,1,1,1\}$. — $5 <= 6$, pick second 5. Remaining balance $6 - 5 = 1$

3. Sorted $\{5,5,5,5,5,2,2,1,1,1\}$. — $5 <= 6$, pick second 5. Remaining balance $6 - 5 = 1$

4. Sorted$\{5,5,5,5,5,2,2,1,1,1\}$. — $1 <= 1$, pick first 1. Remaining balance $1 - 1 = 0$, Done

Solution — [5,5,1]

$$N = \{n_1, n_2, \ldots, n_k\}$$

$$D = \{d_1, d_2, \ldots, d_k\}$$

$$D\_new = [d_1, d_1, \ldots, d_{1_{n_1}}, d_2, d_2, \ldots, d_{2_{n_2}}, \ldots, d_k, d_k, \ldots, d_{k_{n_k}}]$$

$$N = \{n_1, n_2, \ldots, n_k\}$$

$$D = \{d_1, d_2, \ldots, d_k\}$$

$$D\_new = [d_1, d_1, \ldots, d_{1_{n_1}}, d_2, d_2, \ldots, d_{2_{n_2}}, \ldots, d_k, d_k, \ldots, d_{k_{n_k}}]$$

```
D_new = D_new.sort(reverse= True)
S = []
for d in D_new:
      if d ≤ A:
          S.append(d)
          A = A - d
if A == 0:
   Return S
else:
   Print("not possible!")
```

$$N = \{n_1, n_2, \ldots, n_k\}$$

$$D = \{d_1, d_2, \ldots, d_k\}$$

$$D\_new = [d_1, d_1, \ldots, d_{1_{n_1}}, d_2, d_2, \ldots, d_{2_{n_2}}, \ldots, d_k, d_k, \ldots, d_{k_{n_k}}]$$

```
D_new = D_new.sort(reverse= True)
S = []
for d in D_new:
      if d ≤ A:
          S.append(d)
          A = A - d
if A == 0:
   Return S
else:
   Print("not possible!")
```

Time complexity —

$$sorting\ time + Linear\ time\ |D_{new}|$$

Space complexity —-

$$Linear\ time\ |S|$$

$$N = \{n_1, n_2, \ldots, n_k\}$$

$$D = \{d_1, d_2, \ldots, d_k\}$$

$$D\_new = [d_1, d_1, \ldots, d_{1_{n_1}}, d_2, d_2, \ldots, d_{2_{n_2}}, \ldots, d_k, d_k, \ldots, d_{k_{n_k}}]$$

```
D_new = D_new.sort(reverse= True)
S = []
for d in D_new:
      if d ≤ A:
          S.append(d)
          A = A - d
if A == 0:
   Return S
else:
   Print("not possible!")
```

Time complexity —

$$sorting\ time + Linear\ time\ |D_{new}|$$

Space complexity —-

$$Linear\ time\ |S|$$

In the loop — can we stop early ? If yes, when?

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$
2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$
2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$

Greedy says no solution!

Optimal [2,2,2,5]

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$
2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$

Greedy says no solution!

Optimal [2,2,2,5]

Greedy + Backtracking!!!

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $- \ 5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$.

Check $A//min(D) > 0$, continue.

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$. Check $A//min(D) > 0$, continue.

2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$. Check $A//min(D) > 0$, Nope, so backtrack! And choose next best.

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$. Check $A//min(D) > 0$, continue.

2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$. Check $A//min(D) > 0$, Nope, so backtrack! And choose next best.

3. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 6$, pick first 2. Remaining balance $6 - 2 = 4$. Check $A//min(D) > 0$, continue.

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$. Check $A//min(D) > 0$, continue.

2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$. Check $A//min(D) > 0$, Nope, so backtrack! And choose next best.

3. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 6$, pick first 2. Remaining balance $6 - 2 = 4$. Check $A//min(D) > 0$, continue.

4. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 4$, pick second 2. Remaining balance $4 - 2 = 2$. Check $A//min(D) > 0$, continue.

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

1. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 11$, pick first 5. Remaining balance $11 - 5 = 6$. Check $A//min(D) > 0$, continue.

2. Sorted $\{5,5,2,2,2\}$. $-$ $5 <= 6$, pick first 5. Remaining balance $6 - 5 = 1$. Check $A//min(D) > 0$, Nope, so backtrack! And choose next best.

3. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 6$, pick first 2. Remaining balance $6 - 2 = 4$. Check $A//min(D) > 0$, continue.

4. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 4$, pick second 2. Remaining balance $4 - 2 = 2$. Check $A//min(D) > 0$, continue.

5. Sorted $\{5,5,2,2,2\}$. $-$ $2 <= 2$, pick third 2. Remaining balance $2 - 2 = 0$. Done

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

Greedy says no solution!

Optimal [2,2,2,5]

Is greedy always optimal?

$D = \{2,5\}, A = 11$, 3 copies of 2, 2 copies 5.

$D_{new} = [2,2,2,5,5]$

Greedy says no solution!

Optimal [2,2,2,5]

Greedy + Backtracking — a clever way of trying all combinations .

Now if we don’t know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

Now if we don’t know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)
while balance != 0:
     flag = 1
      for d in D:
         if d <= balance:
              flag = 0
              change.append(d)
              balance -= d
              break
     if flag:
         raise ValueError("not possible”)
# solution is change
```

Now if we don’t know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)
while balance != 0:
     flag = 1
      for d in D:
         if d <= balance:
              flag = 0
              change.append(d)
              balance -= d
              break
     if flag:
         raise ValueError("not possible”)
# solution is change
```

Does it work?

Now if we don't know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)
while balance != 0:
     flag = 1
      for d in D:
         if d <= balance:
              flag = 0
              change.append(d)
              balance -= d
              break
     if flag:
         raise ValueError("not possible”)
# solution is change
```

Does it work?

$$D = \{1,2,5\}, A = 11$$

Now if we don't know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)
while balance != 0:
     flag = 1
      for d in D:
         if d <= balance:
              flag = 0
              change.append(d)
              balance -= d
              break
     if flag:
         raise ValueError("not possible”)
# solution is change
```

Does it work?

$D = \{1,2,5\}, A = 11$

This is greedy (without backtracking)

Now if we don't know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)
while balance != 0:
     flag = 1
      for d in D:
         if d <= balance:
              flag = 0
              change.append(d)
              balance -= d
              break
     if flag:
         raise ValueError("not possible”)
# solution is change
```

Does it work?

$D = \{1,2,5\}, A = 11$

This is greedy (without backtracking)

Can we improve it?

Now if we don't know how many copies each coin has? We are allowed to have as many copies as required.
The goal is to use minimum number of demonizations.
Input D, A

```
balance, change = A, []
D.sort(reverse=True)

while balance != 0:
    for d in D:
        if d <= balance:
            k = balance // d
            change.extend([d]*k)
            balance -= d*k
            break
    else:
        return None
return change
```

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

```
N = A + B
N.sort()
```

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

```
N = A + B
N.sort()
```

What if you are not allowed to use sort — give us an algorithm to merge two sorted list A and B such that the result is also sorted.

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

```
N = A + B
N.sort()
```

What if you are not allowed to use sort — give us an algorithm to merge two sorted list A and B such that the result is also sorted.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

Moving on!!

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

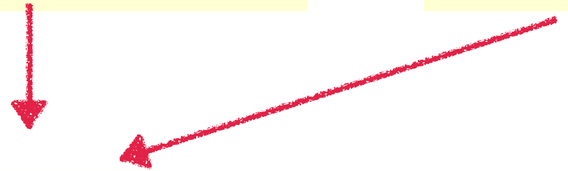
Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```



$C = [2]$

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

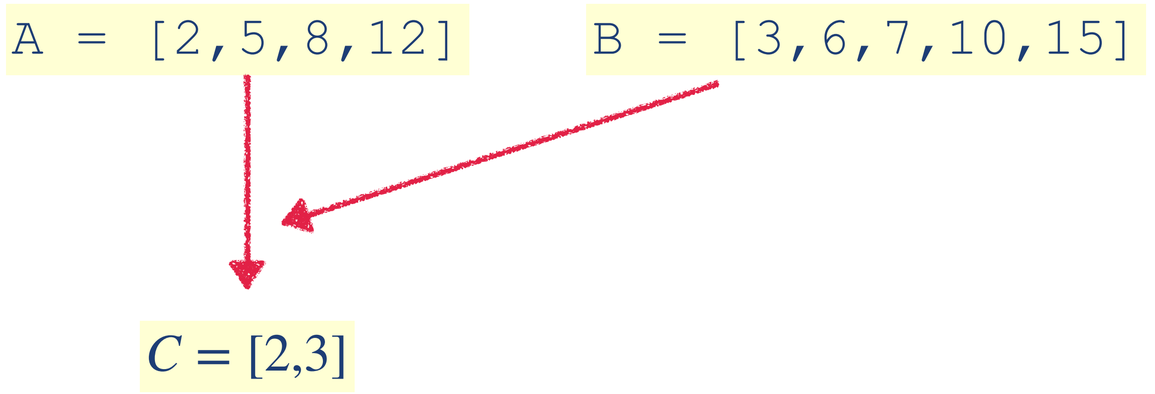
```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.



```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

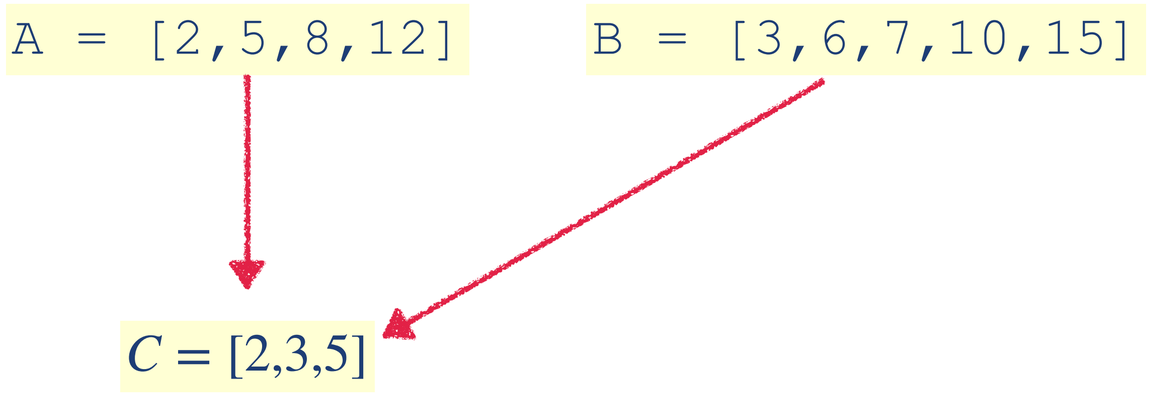
```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.



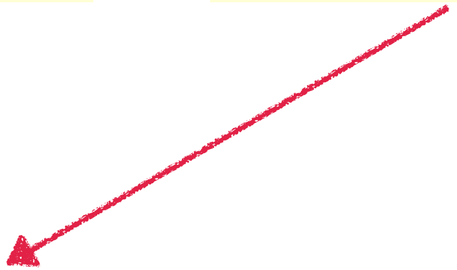
```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```



Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

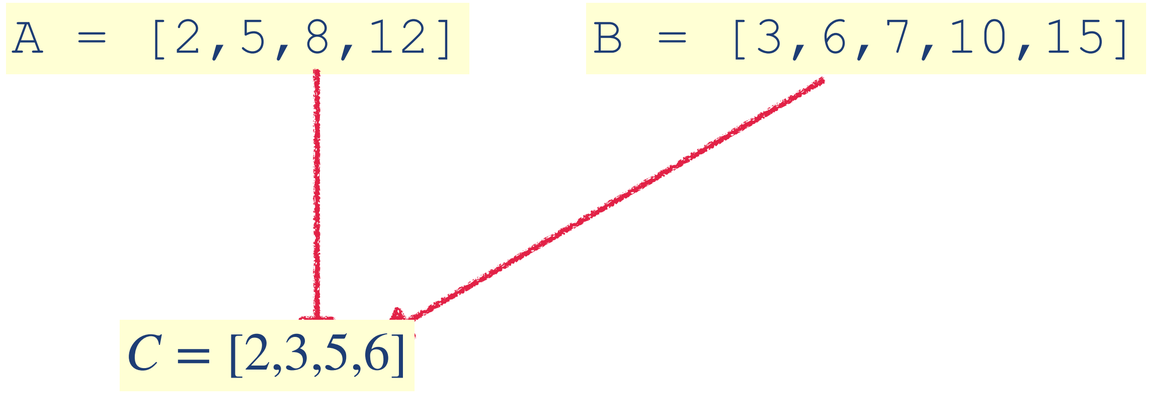
```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.



```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

```
A = [2,5,8,12]
```

```
B = [3,6,7,10,15]
```

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$**or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
C = [2,3,5,6,7,8,10,12,15]
```

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

```
C = [2,3,5,6,7,8,10,12,15]
```

Intuitive Idea

Since both A and B are already sorted:

- The **smallest element** among all will be **either** $A[0]$ **or** $B[0]$.
- Once we take the smaller of the two, we move **forward in that list**.
- We repeat this comparison and selection process until both lists are fully processed.

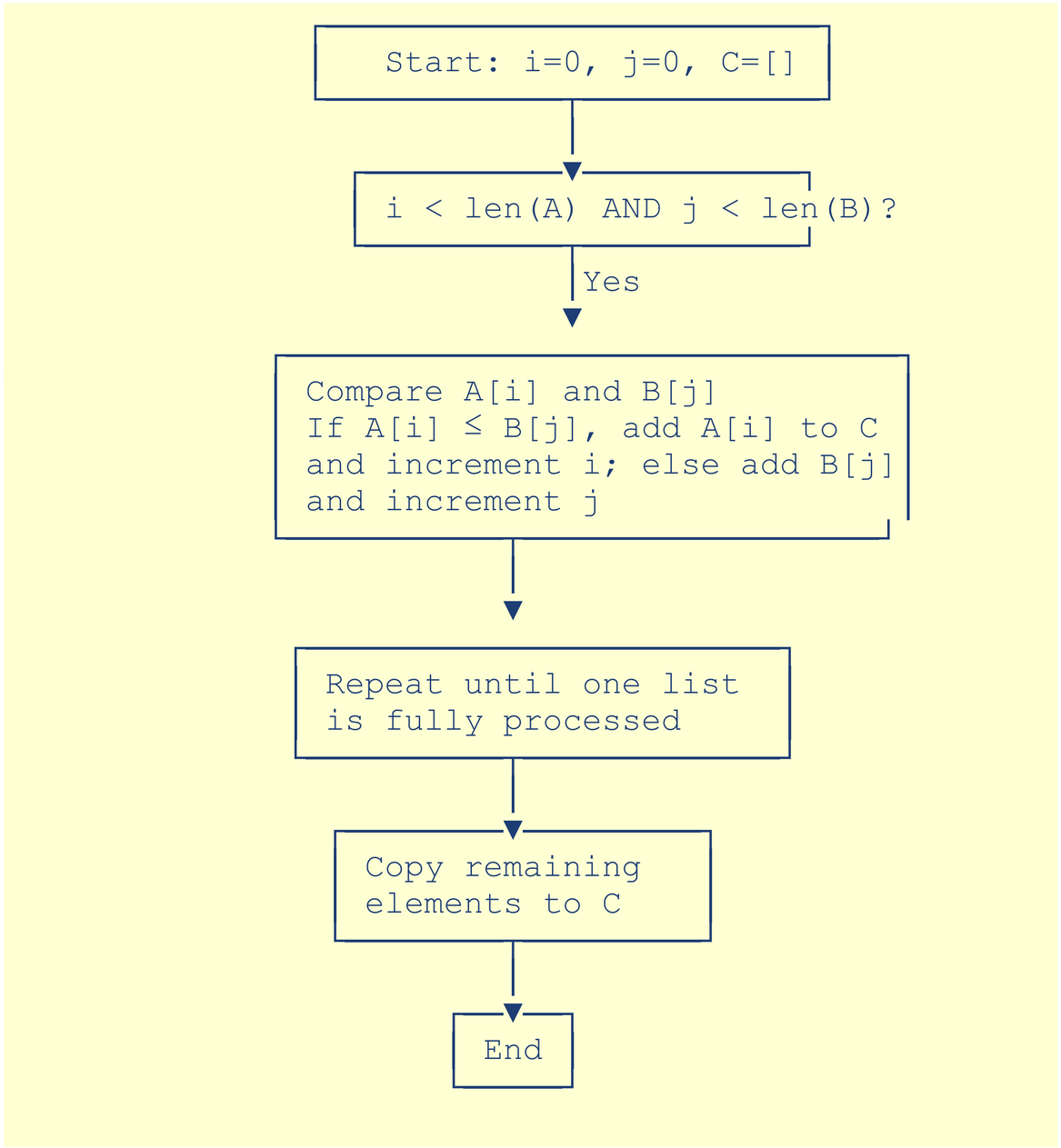
Step	A pointer	B pointer	Smaller element	Output C
1	2	3	2 (from A)	[2]
2	5	3	3 (from B)	[2, 3]
3	5	6	5 (from A)	[2, 3, 5]
4	8	6	6 (from B)	[2, 3, 5, 6]
5	8	7	7 (from B)	[2, 3, 5, 6, 7]
6	8	10	8 (from A)	[2, 3, 5, 6, 7, 8]
7	12	10	10 (from B)	[2, 3, 5, 6, 7, 8, 10]
8	12	15	12 (from A)	[2, 3, 5, 6, 7, 8, 10, 12]
9	—	15	15 (remaining from B)	[2, 3, 5, 6, 7, 8, 10, 12, 15]

```
A = [2, 5, 8, 12]
B = [3, 6, 7, 10, 15]
```

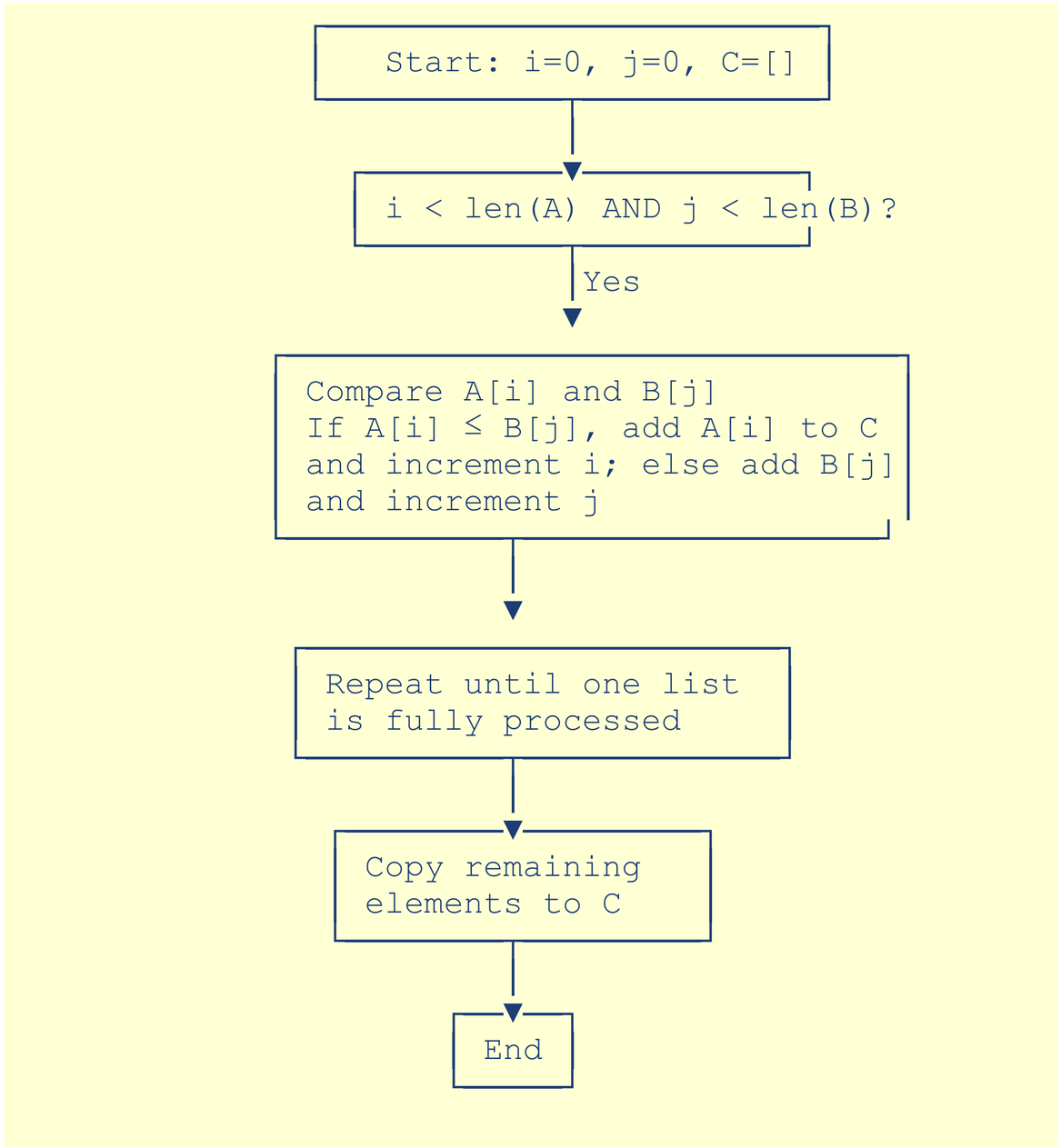
```
C = [2,3,5,6,7,8,10,12,15]
```

```
Initialize indices:
Set three pointers —
i = 1 (for list A), j = 1 (for list B), k = 1 (for list C).
Compare and copy:
While both i ≤ m and j ≤ n:
        If A[i] ≤ B[j], then
                set C[k] ← A[i], and increment i and k.
        Else
                set C[k] ← B[j], and increment j and k.
Copy remaining elements:
      If list A still has elements (i.e., i ≤ m),
                copy all remaining elements of A into C.
      if list B still has elements (j ≤ n),
                 copy all remaining elements of B into C.
The array C now contains all elements of A and B in sorted
order.
```

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B



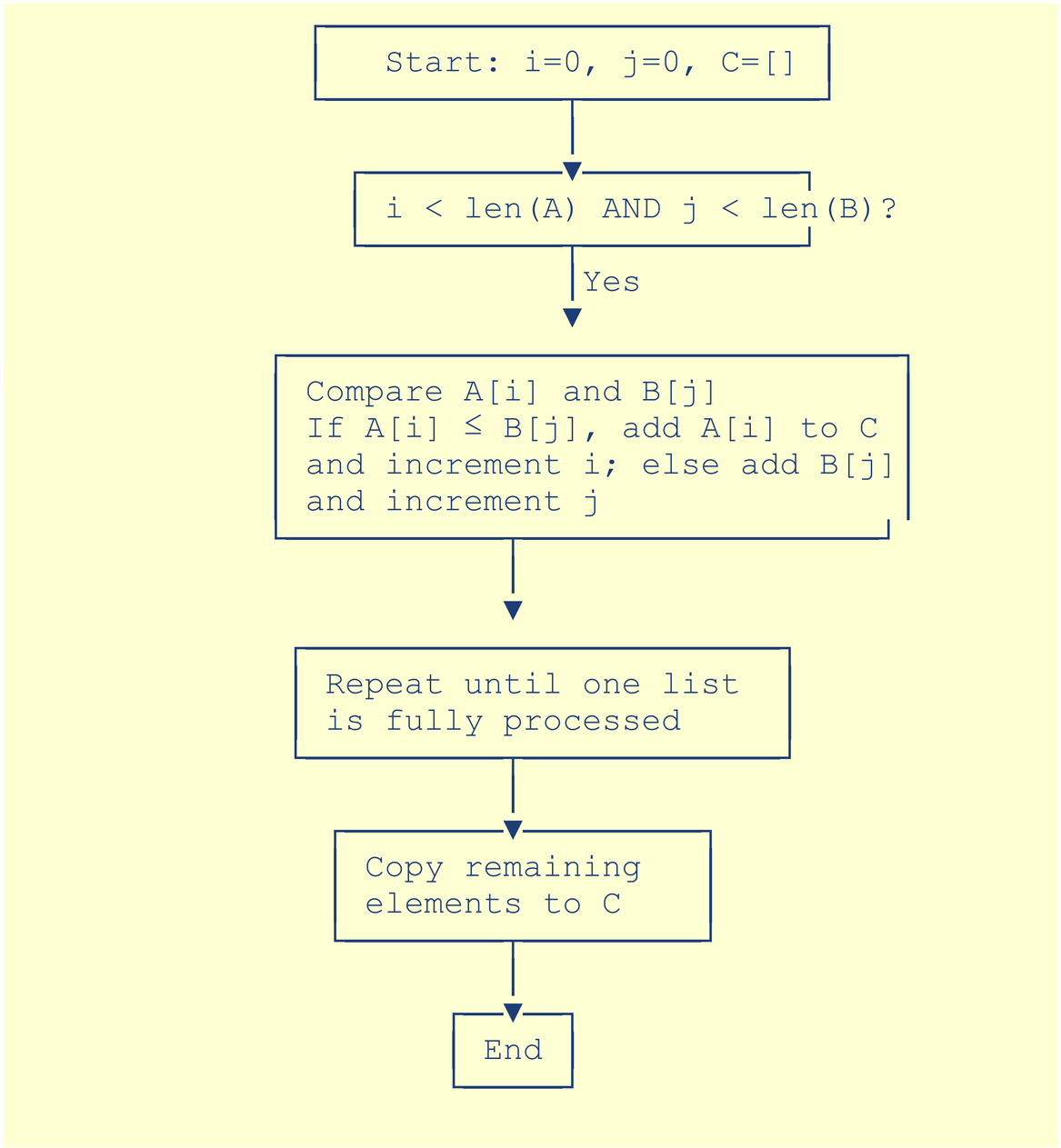
Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B



Time Complexity

$$|A| + |B|$$

Given — two sorted list (A,B)
Output — A final sorted list of merged list A+B



Time Complexity

$$|A| + |B|$$

Space Complexity

$$|A| + |B|$$

Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

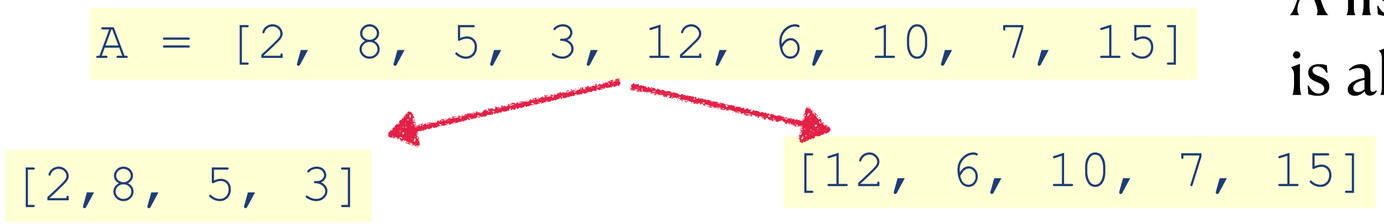
```
A = [2, 8, 5, 3, 12, 6, 10, 7, 15]
```

Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

```
A = [2, 8, 5, 3, 12, 6, 10, 7, 15]
```

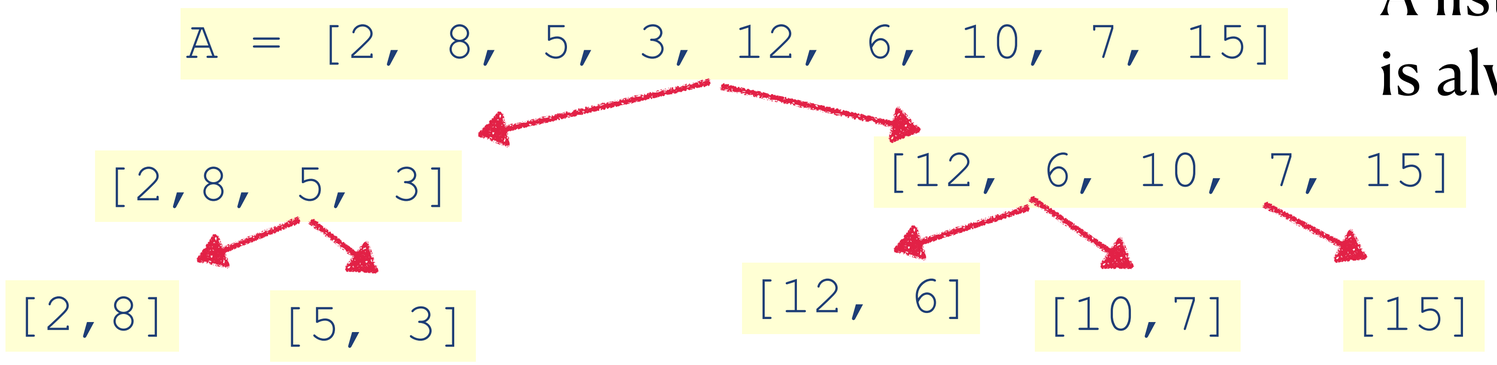
A list with single element is always sorted

Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?



A list with single element is always sorted

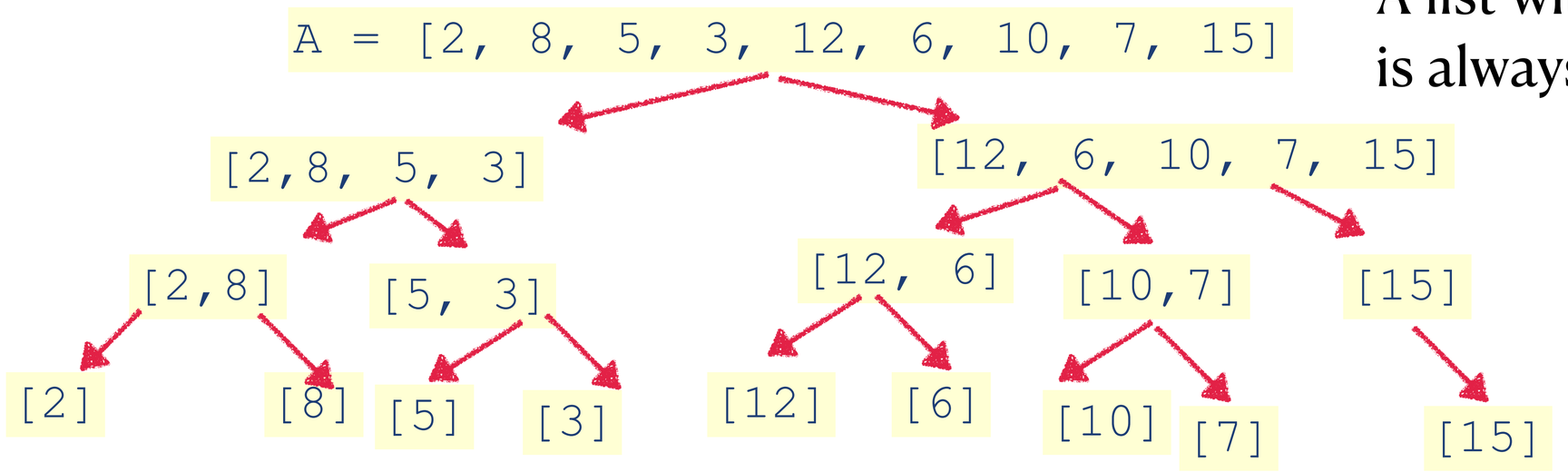
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?



A list with single element is always sorted

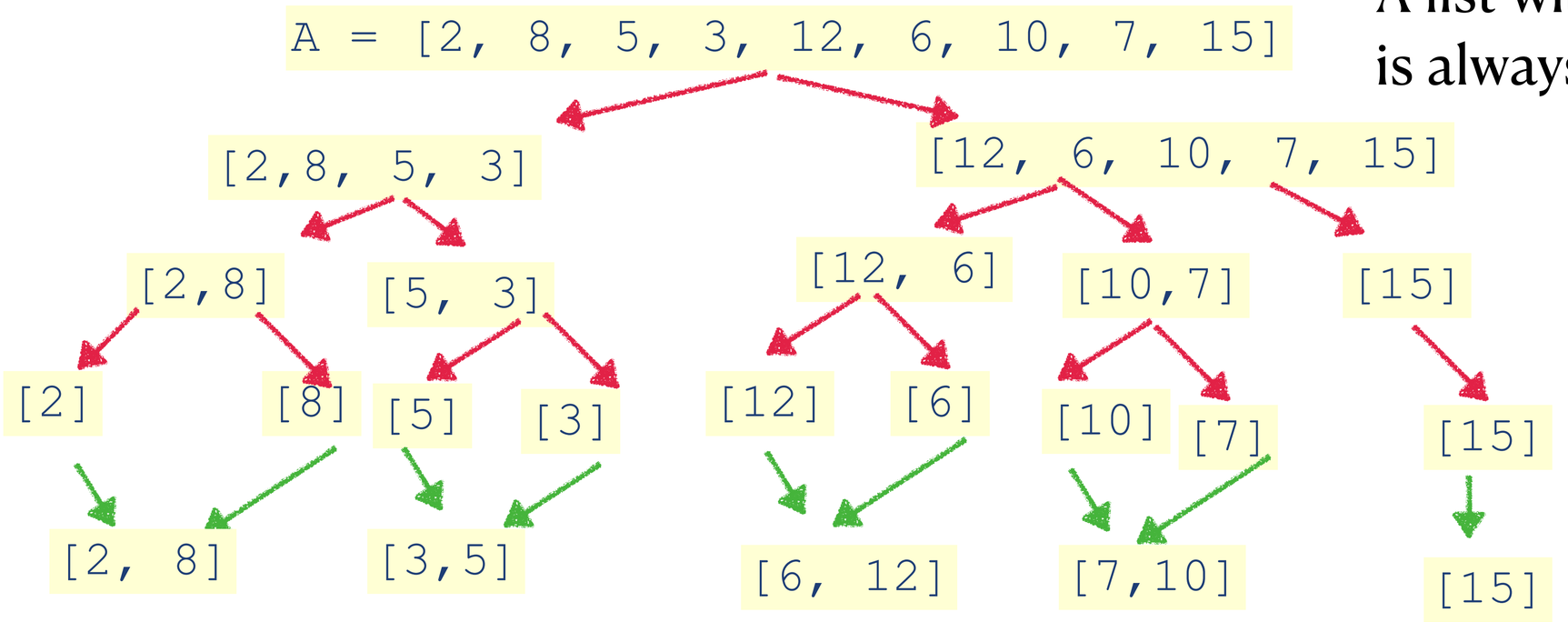
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



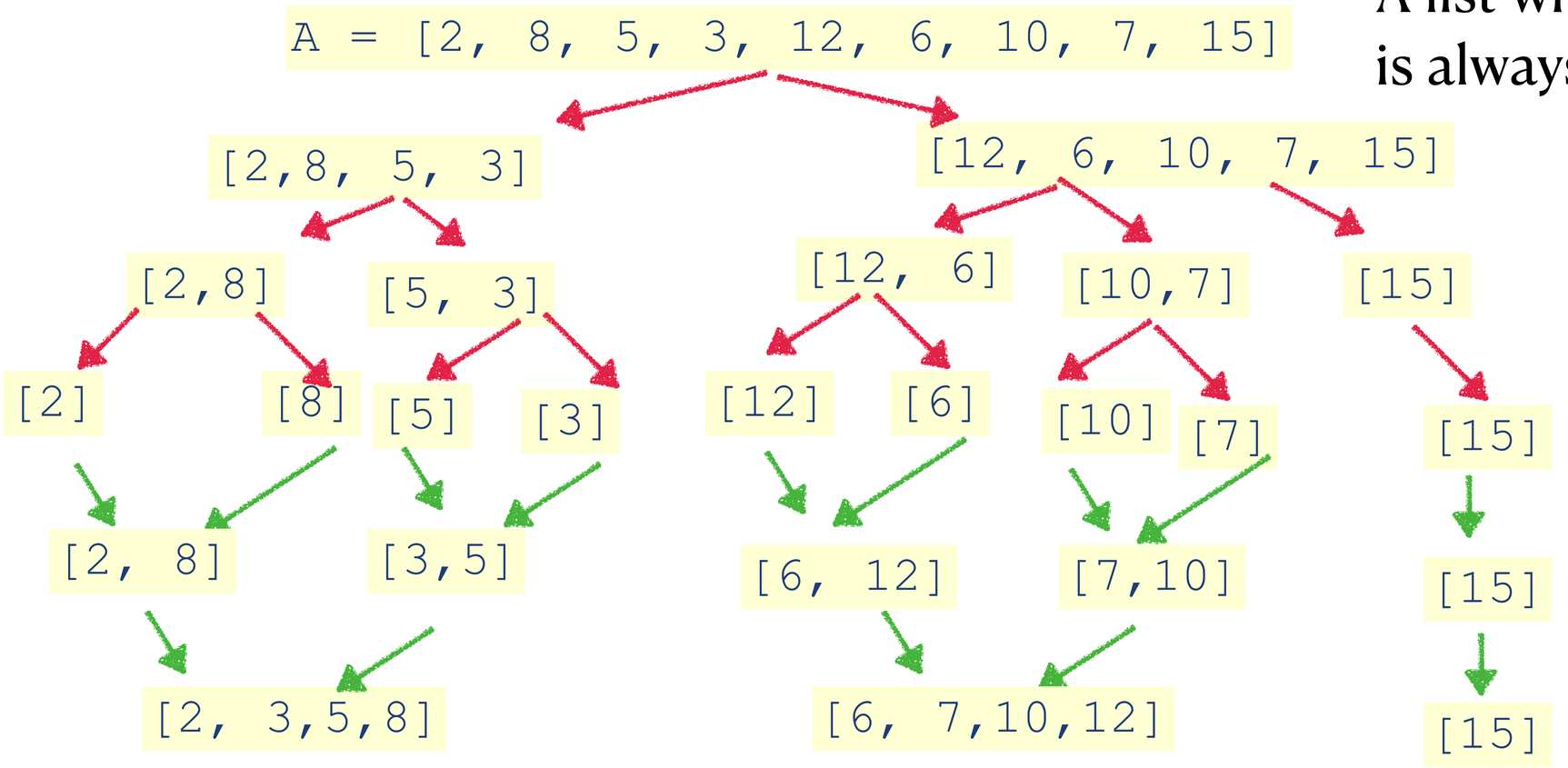
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



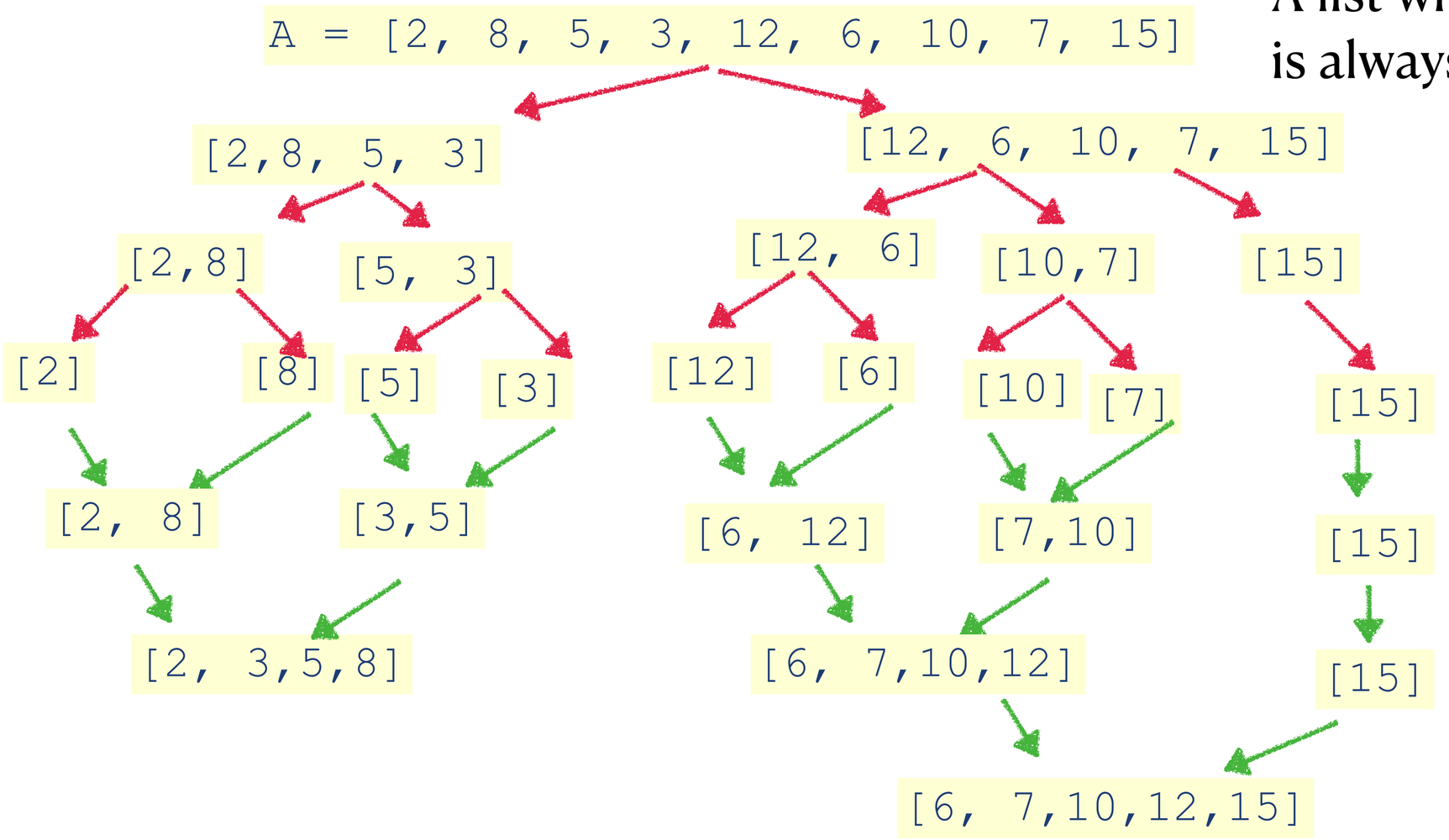
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



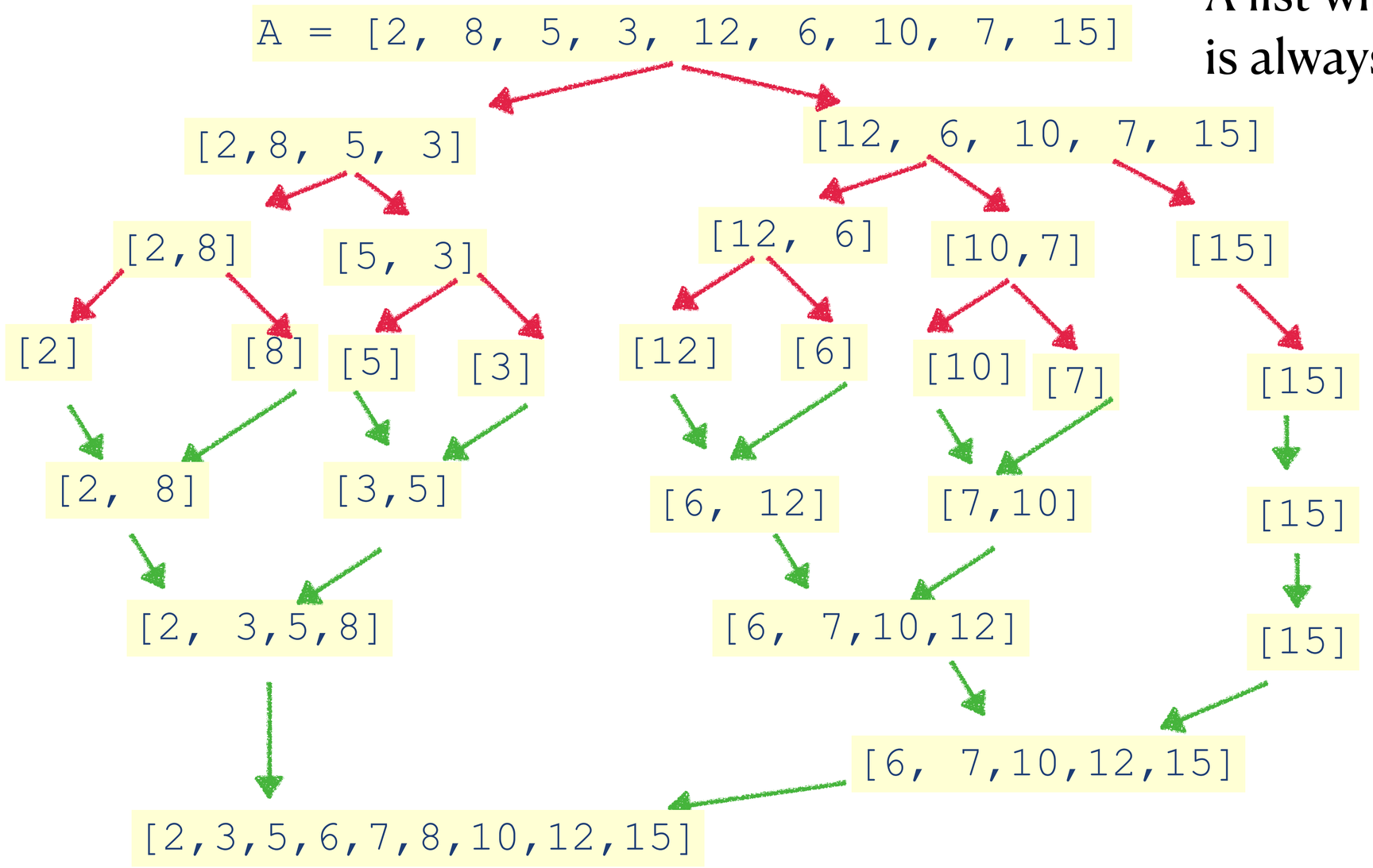
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



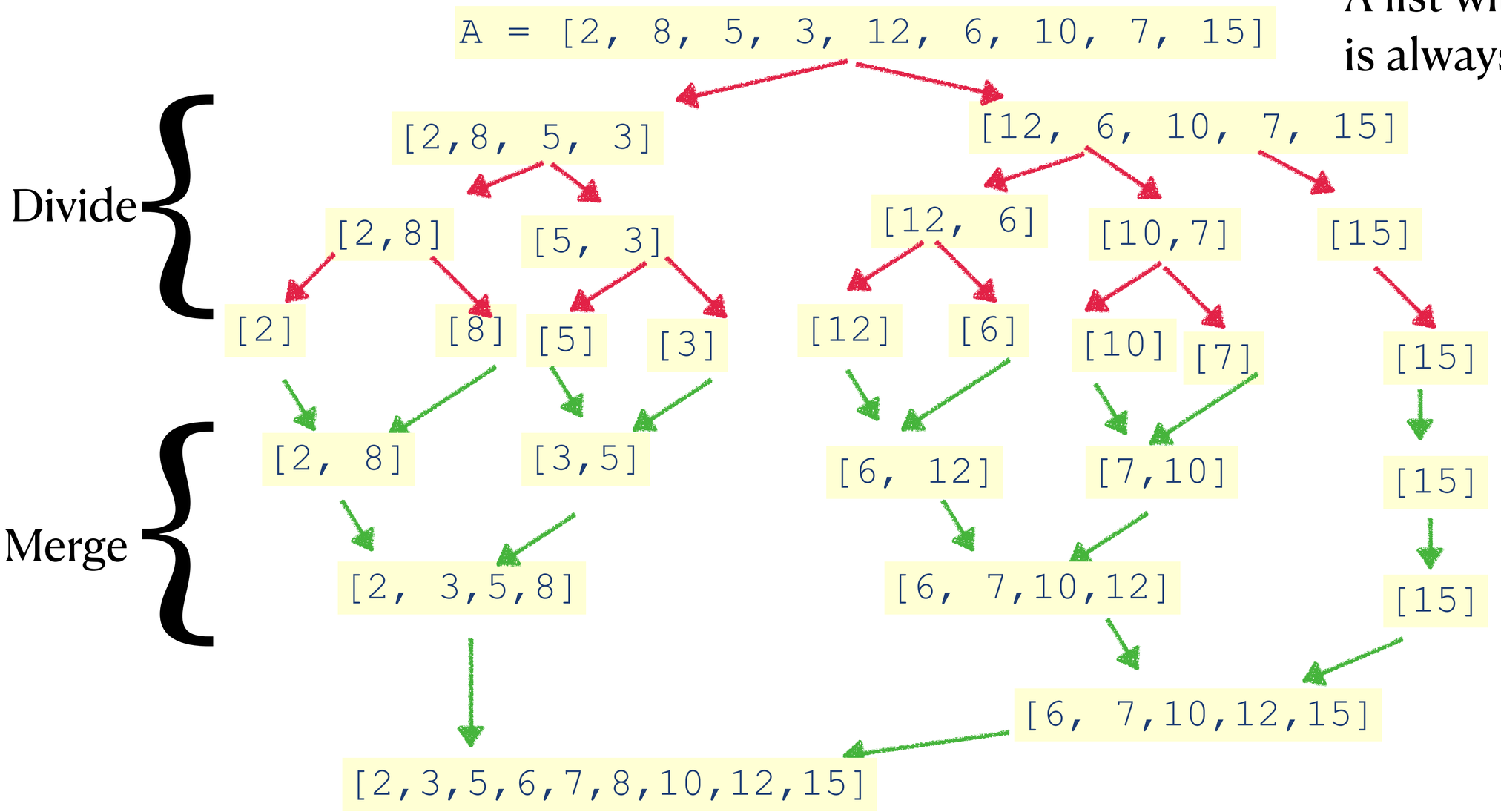
Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



Can you use the idea of merging two sorted lists to develop an algorithm for sorting a given list?

A list with single element is always sorted



Algorithm: Divide and Conquer

Three steps of the Divide and Conquer method:

Divide: Break the problem into smaller subproblems.

Conquer: Solve the subproblems recursively.

Combine: Merge or join the sub-results to form the final solution.

Algorithm: Divide and Conquer

Three steps of the Divide and Conquer method:

Divide: Break the problem into smaller subproblems.

Conquer: Solve the subproblems recursively.

Combine: Merge or join the sub-results to form the final solution.

Step	Sorting Algorithm we just discussed
Divide	Split the list into halves
Conquer	Recursively sort halves
Combine	Merge the sorted halves

Algorithm: Divide and Conquer

Three steps of the Divide and Conquer method:

Divide: Break the problem into smaller subproblems.

Conquer: Solve the subproblems recursively.

Combine: Merge or join the sub-results to form the final solution.

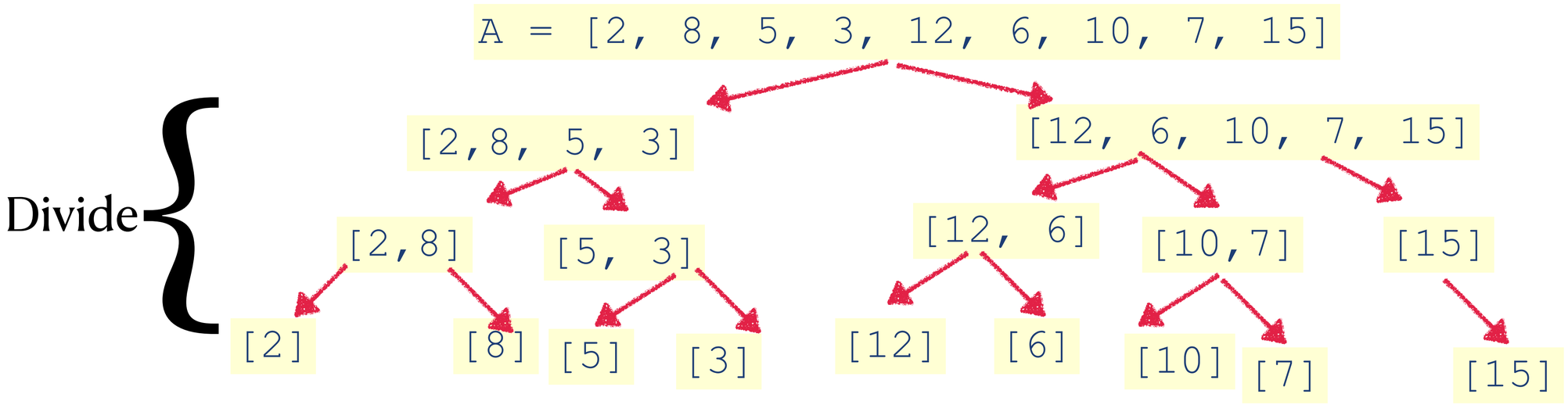
Step	Sorting Algorithm we just discussed
Divide	Split the list into halves
Conquer	Recursively sort halves
Combine	Merge the sorted halves

} Merge Sort

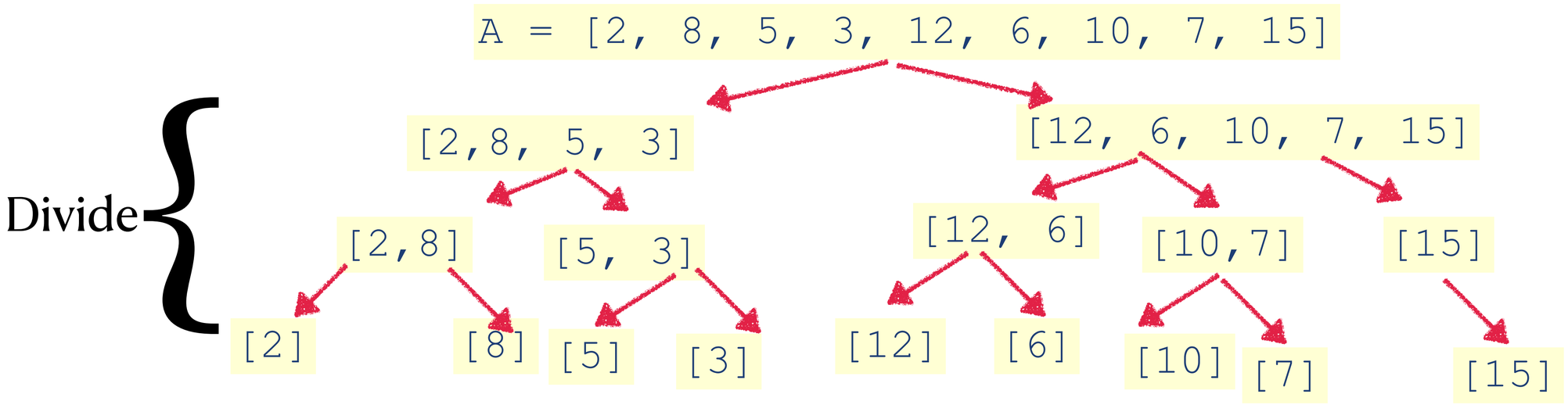
Every time Merge Sort runs, it performs:

Division (Splitting) — repeatedly divides the list into halves.

Merging (Combining) — merges the smaller lists back into a sorted whole.

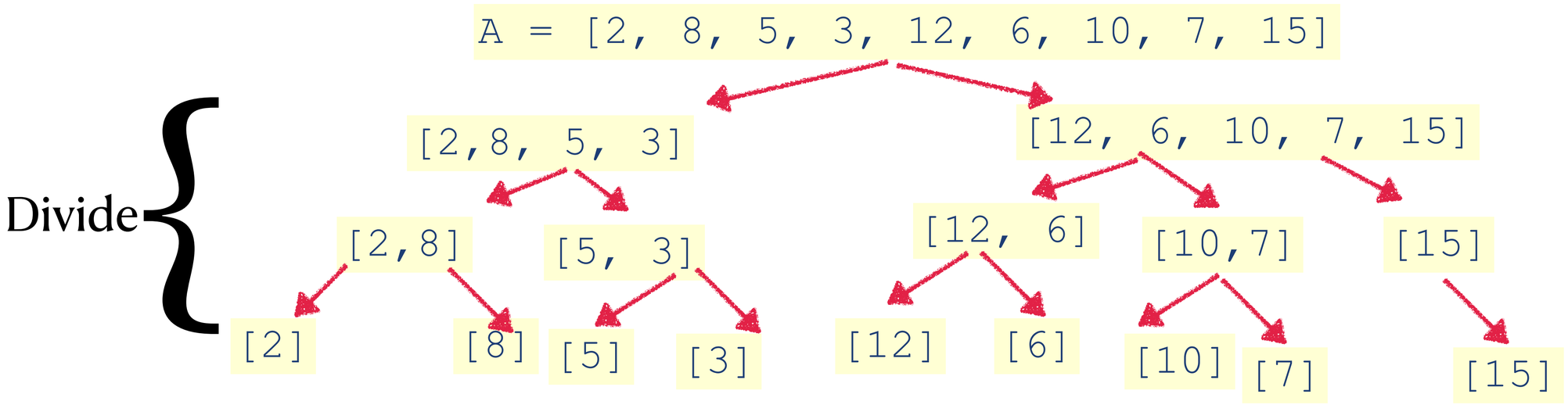


Every time, we are diving the list into half until a single element is there!
How many times we have to do?



Every time, we are diving the list into half until a single element is there!
How many times we have to do?

$$\frac{n}{2^k} = 1 \longrightarrow n = 2^k \longrightarrow k = log_2(n)$$



Every time, we are diving the list into half until a single element is there!
How many times we have to do?

$$\frac{n}{2^k} = 1 \longrightarrow n = 2^k \longrightarrow k = log_2(n)$$

Total $log_2 n$ many levels

Every time Merge Sort runs, it performs:

Division (Splitting) — repeatedly divides the list into halves. — $log_2 n$
Merging (Combining) — merges the smaller lists back into a sorted whole.

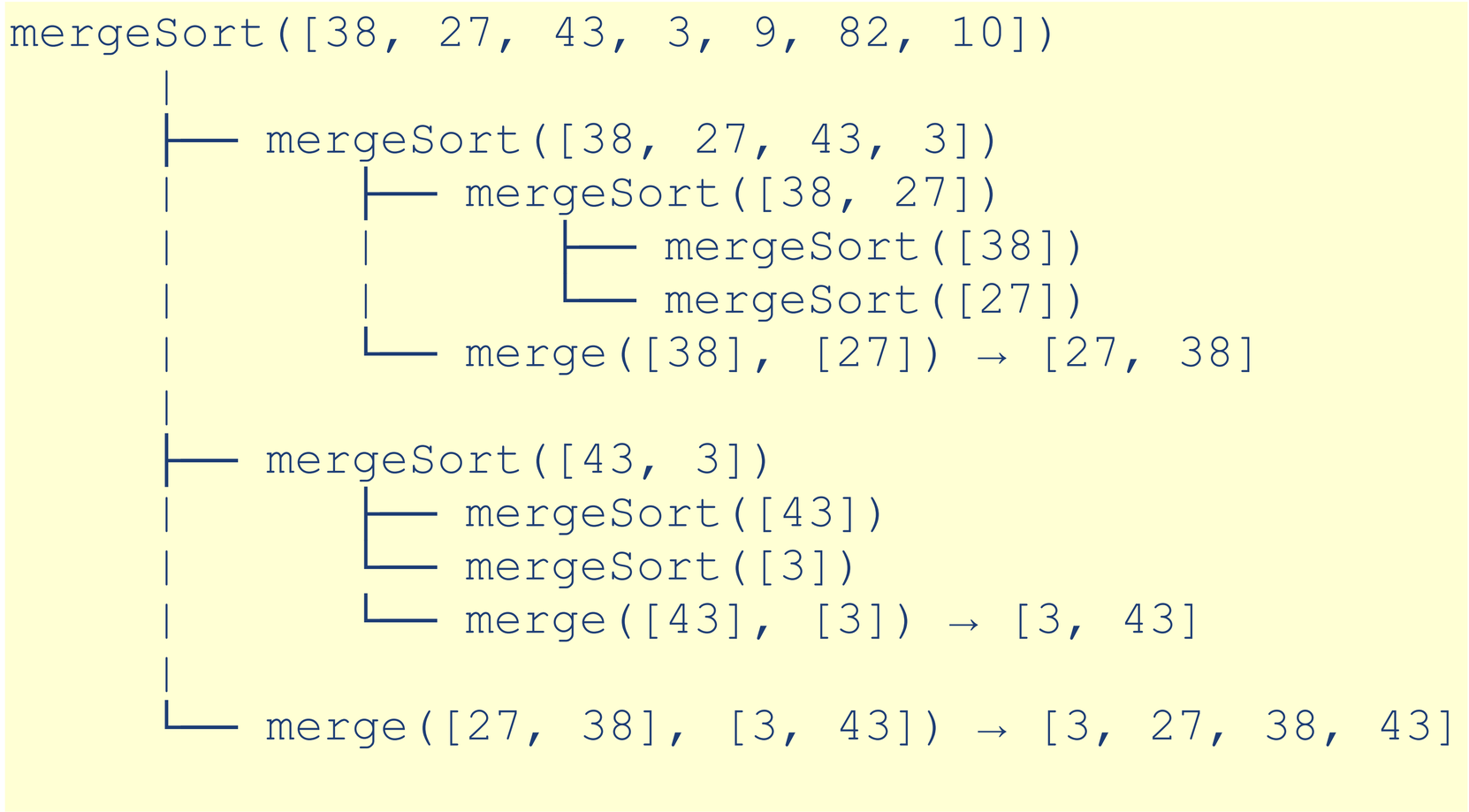
```
Now look at the merging:
At each level of recursion
     we merge sublists.
     every element participates in exactly one merge per level.

So, at any given level, total work = O(n) (because every
element is compared and copied once).

total number of level = log n
Therefore, merging = n \times log n
```

Total Merge Sort — $logn + n \times logn$

```
ALGORITHM MERGE_SORT(A, low, high)
    // A is the array to be sorted
    // low and high are the indices of the first and last elements
if low < high then
       mid ← ⌊(low + high) / 2⌋          // find middle index
       MERGE_SORT(A, low, mid)           // sort first half
       MERGE_SORT(A, mid + 1, high)      // sort second half
       MERGE(A, low, mid, high)          // merge the two halves
```

```
ALGORITHM MERGE(A, low, mid, high)
    Merges two sorted subarrays: Left part: A[low .. mid] Right part: A[mid+1 .. high]
n1 ← mid - low + 1                     // length of left part
n2 ← high - mid                        // length of right part
 Create arrays L[1 .. n1] and R[1 .. n2]
 for i ← 1 to n1 do
       L[i] ← A[low + i - 1]
 for j ← 1 to n2 do
       R[j] ← A[mid + j]
 i ← 1, j ← 1, k ← low                  // initial indices
 while i ≤ n1 and j ≤ n2 do
       if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
       else
            A[k] ← R[j]
            j ← j + 1
       k ← k + 1
while i ≤ n1 do
        A[k] ← L[i]
        i ← i + 1
        k ← k + 1
while j ≤ n2 do
       A[k] ← R[j]
        j ← j + 1
       k ← k + 1
```

Merge Sort!


mergeSort([38, 27, 43, 3, 9, 82, 10])
mergeSort([38, 27, 43, 3])
mergeSort([38, 27])
mergeSort([38])
mergeSort([27])
merge([38], [27]) → [27, 38]
mergeSort([43, 3])
mergeSort([43])
mergeSort([3])
merge([43], [3]) → [3, 43]
merge([27, 38], [3, 43]) → [3, 27, 38, 43]