BİL3014

Algoritma Analizi

Dr. Öğr. Üyesi Emre DELİBAŞ

Recursive Algoritmalar ve Recurrence Bağıntısı

Rekurans Bağıntısı

- Bir dizinin içinde kendisinden bir parça bulunuyorsa o diziye rekurans bağıntısı (özyineli dizi) denir

$a_n = 5n+1 \Longrightarrow a_{20} = 101$ (dizi) sequence

$\boxed{a_n = 2a_{n-1} + 3}$ (a_{n-1} kendisinden parçadır) Rekürans Bağıntısı

$\downarrow$

$a_0 = 3, \; n \geq 1$

$$\left.\begin{aligned} a_n &= a_{n-1} - 2a_{n-2} + 5^n \\ a_n &= 3a_{n-1} + 2a_{n-2} \end{aligned}\right\} \text{rek. bağıntısı} \quad \longrightarrow a_1 = 3, \; a_0 = 5, \; n \geq 2$$

Rekurans Bağıntısı

Örn: $a_n = 2a_{n-1} - a_{n-2} + 2^{n-1}$, $a_0 = 1$, $a_1 = 2$, $n \geq 2$ olarak veriliyor. Buna göre $a_5 = ?$

$n = 2 \Rightarrow a_2 = 2a_1 - a_0 + 2 = 4 - 1 + 2 \Rightarrow \boxed{a_2 = 5}$

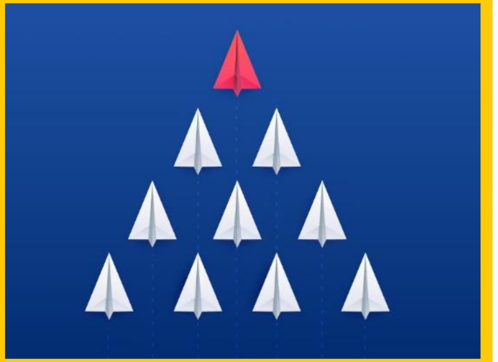
$n = 3 \Rightarrow a_3 = 2a_2 - a_1 + 4 = 10 - 2 + 4 \Rightarrow \boxed{a_3 = 12}$

$n = 4 \Rightarrow a_4 = 2a_3 - a_2 + 8 = 24 - 5 + 8 \Rightarrow \boxed{a_4 = 27}$

$n = 5 \Rightarrow a_5 = 2a_4 - a_3 + 16 = 54 - 12 + 16 \Rightarrow a_5 = 58$

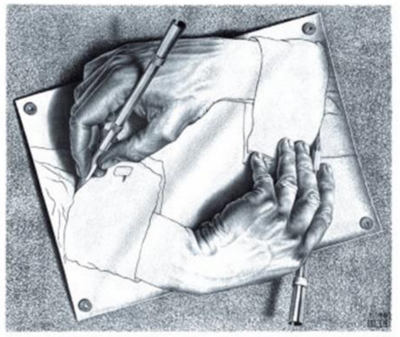
Böl-Yönet Yaklaşımı

- Problem çözümlemede problemin daha küçük alt problemlere parçalanarak çözümlenmesine **böl ve yönet (devide and conquer) yaklaşımı** denir.
- Problem kendisine benzer küçük boyutlu alt problemlere bölünür.
- Alt problemler çözülür ve bulunan çözümler birleştirilir.
    - **Divide:** Problem iki veya daha fazla alt probleme bölünür
    - **Conquer:** Divide-and-conquer özyineleme (recursively) kullanılarak alt problemleri çözer.
    - **Combine:** Alt problemlerin çözümleri alınır ve orijinal problemin çözümü olacak şekilde birleştirilir.

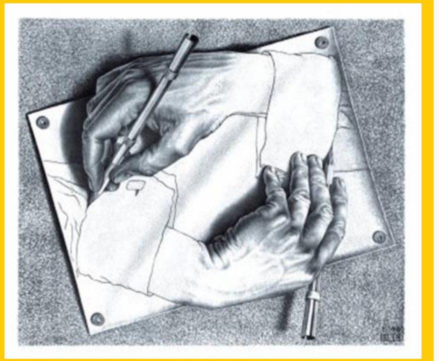


Özyineleme (Recursion)

- Özyineleme bir metodun kendisini çağırdığı bir programlama tekniğidir.
- Birçok algoritmanın uygulanmasında özyinelemeli yaklaşımın uygulanabilirliği söz konusudur.
- Genellikle bu yöntem ***n*** boyutlu problemin daha küçük boyutları ele alınarak benzeri şekilde çözümünde kullanılmaktadır.



Özyineleme (Recursion)



- Özyinelemenin sonsuz olarak devam etmesinin önlenmesi için bitiş koşulunun olması gerekmektedir.
- Dolayısıyla özyinelemeli programlarda bu sürecin ne zaman veya hangi koşullarda durması gerektiği önceden bilinmelidir.
- Tanım gereği fonksiyon kendi kendini çağırdığında genellikle her seferinde daha küçük boyutlu problem ele alınacağından sürecin sonsuz devam edemeyeceği anlaşılmaktadır.
- Özyineleme işlemi durdurma durumu sağlanınca sonlandırılır.

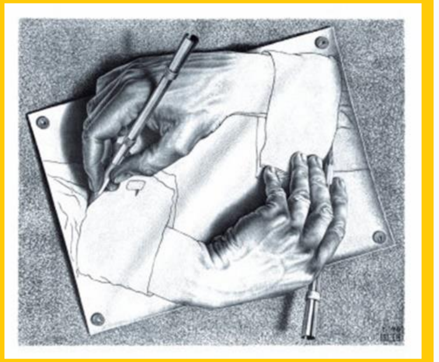
Özyineleme (Recursion)

- Çarpımı toplamla ifade ettiğimizde veya toplamı ardışık olarak 1 ler şeklinde gösterdiğimizde rekursifliğe başvurmaktayız:

6 * 5 = (6 * 4) + 6 = ((6 * 3) + 6) + 6 = (((6 * 2) + 6) + 6) + 6
= ((((6 * 1) + 6) + 6) + 6) + 6 = 6 + 6 + 6 + 6 + 6 = 30

- Veya

6 + 5 = (6 + 4) + 1 = ((6 + 3) + 1) + 1) = (((6 + 2) + 1) + 1) + 1
= ((((6 + 1) + 1) + 1) + 1) + 1 = 6 + 1 + 1 + 1 + 1 + 1 = 11

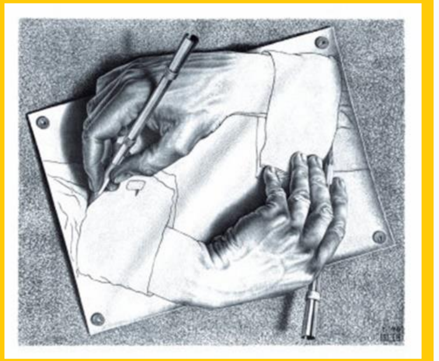


Özyineleme (Recursion)

- Özyinelemeli algoritmalar anlaşıldığı gibi iki temel kısımdan oluşmaktadır. Bunlar;
- Algoritmanın sonlandırılmasını sağlayan ***bitiş kısmı (base case)*** ve,
- Fonksiyonun belirtilen sayıda tekrarlanmasını sağlayan ***gövde kısmı (recursive case)*** dır.

- Genel yazımı:

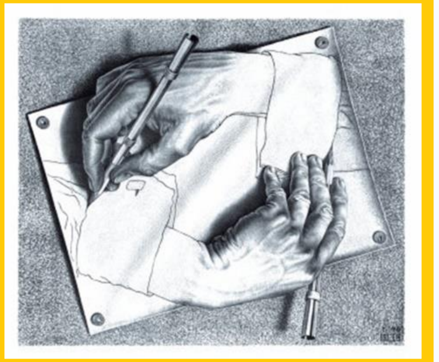
```
if (durdurma durumu sağlandıysa)
        çözümü yap
else
        problemi özyineleme kullanarak indirge
```



Özyineleme (Recursion)

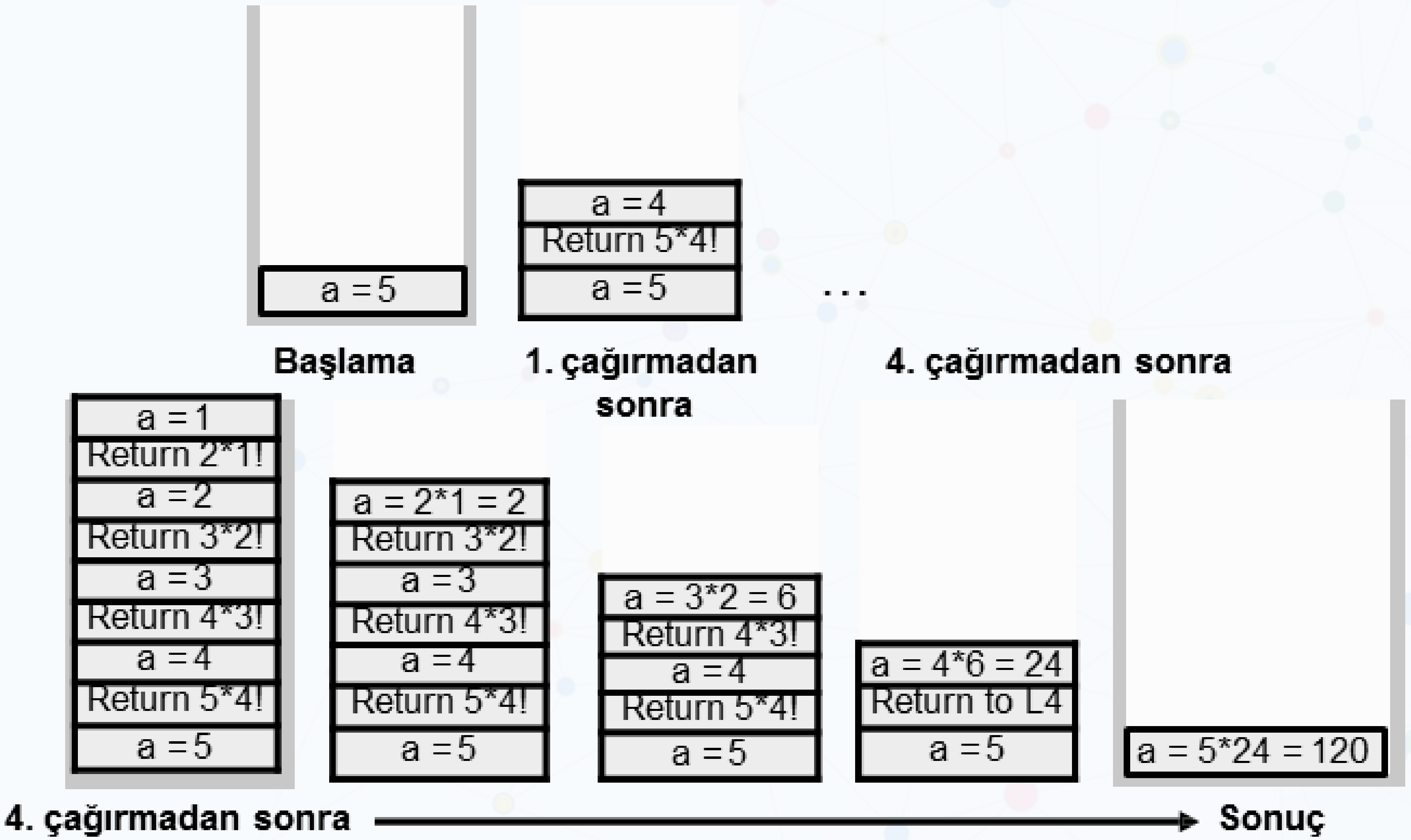
- **Faktöriyelin özyineli olarak hesaplanması**
- 0! = 1 olduğundan n >= 1 ise n! = n * (n - 1)!
- Burada 0! = 1 bitiş koşulu olmaktadır. Fakat bu koşul programcıya bağlı olarak örneğin 2! = 2 şeklinde de değiştirilebilir.

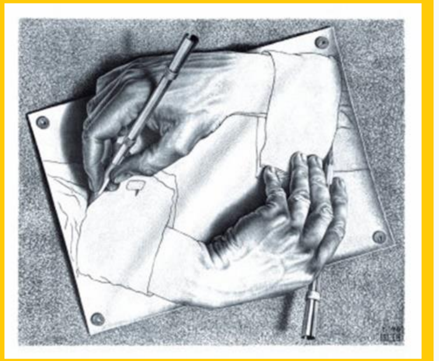
```
int faktoriyel(int n){
        if (n==0)
                return 1;
        return n * faktoriyel(n-1);
}
```



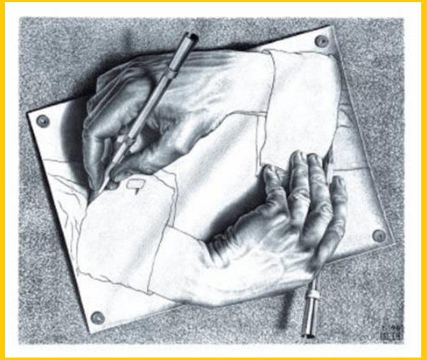
Özyineleme (Recursion)

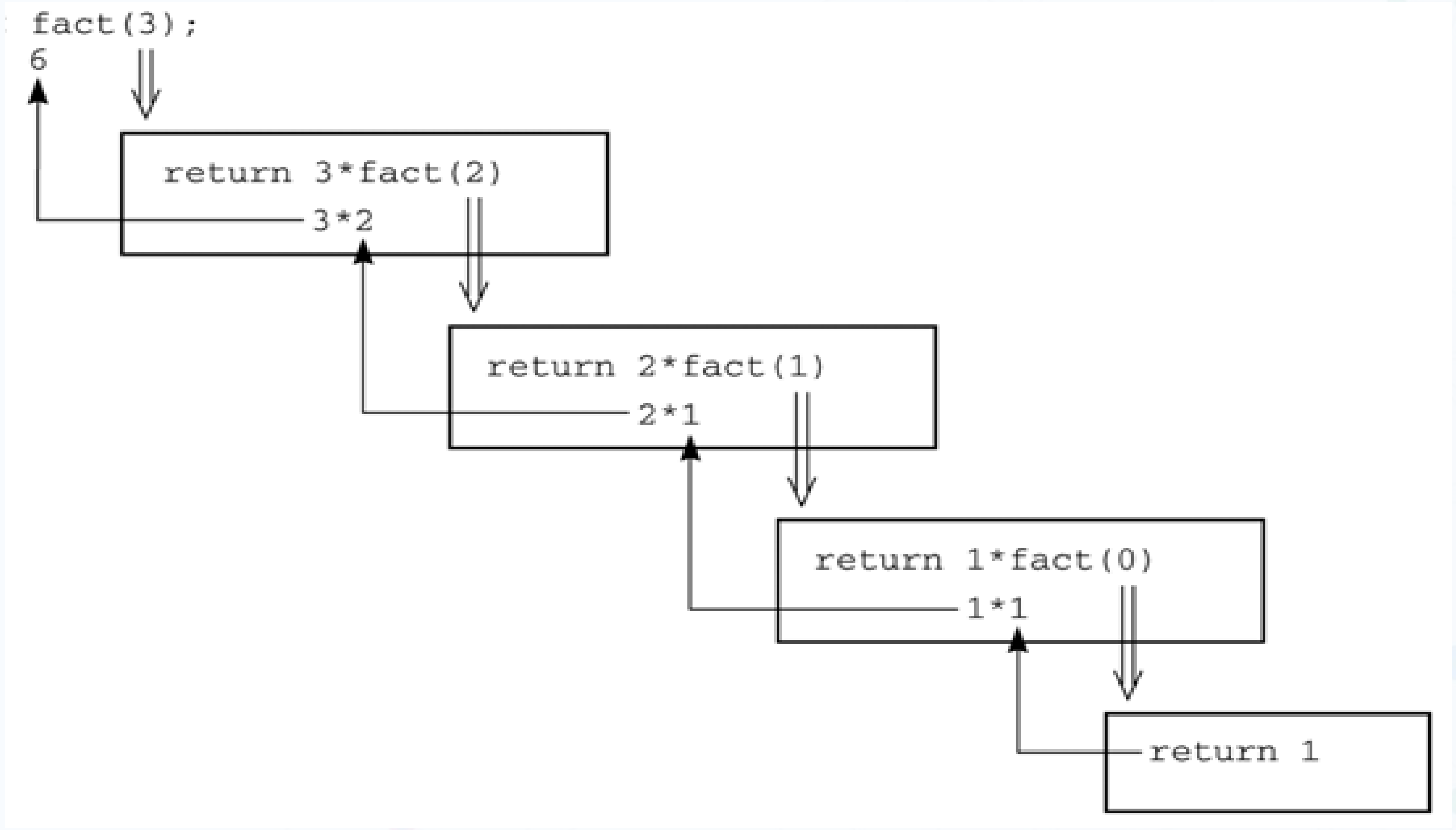
- faktoriyel(5) için çalışmasının stack ile gösterimi.





Özyineleme (Recursion)





Özyineleme (Recursion)

- Özyineleme programlamada bazı algoritmaların anlaşılmasında kolaylıklar sağlasa da birçok durumlarda olumsuz yönlerini de göstermektedir.
- Devamlı olarak fonksiyonun kendi kendini çağırması belirli işlem yüküne de sebep olacaktır.
- Çünkü her fonksiyonun tekrar çağırılması birçok atama işleminin gerçekleştirilmesinin ve yeni değişkenlere uygun bölge ayırımını gerektirmektedir.
- Bu nedenle aynı özyinelemede süreç kendisini çok sayıda çağırdığından bellek ve zaman problemi ortaya çıkacaktır.
- Dolayısıyla eğer programın özyinelemesiz çözümü mümkünse bu çözüm biçimi tercih edilebilir.

Insertion Sort

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

O(n) work per iteration

O(n) iterations

Total work done:

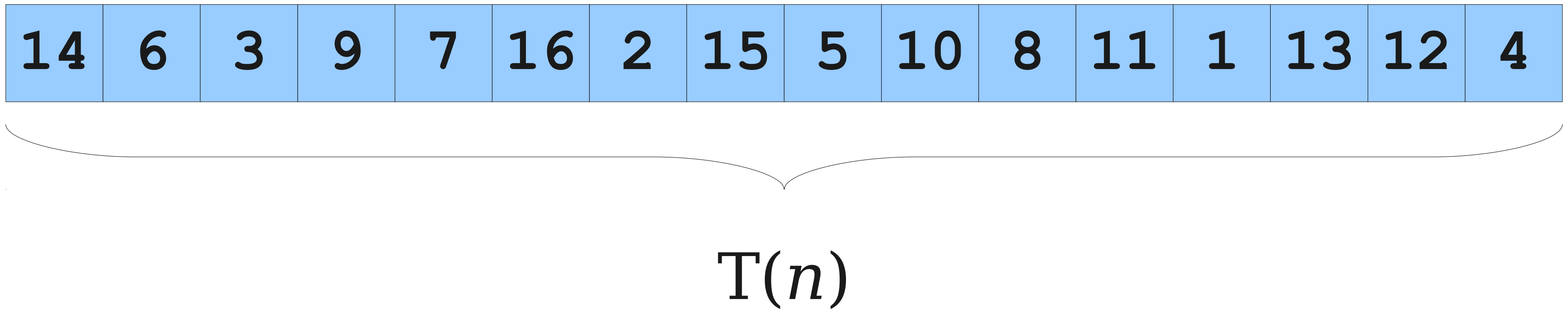
O(n^2)

Daha iyi bir sıralama algoritması olarak; **Mergesort**

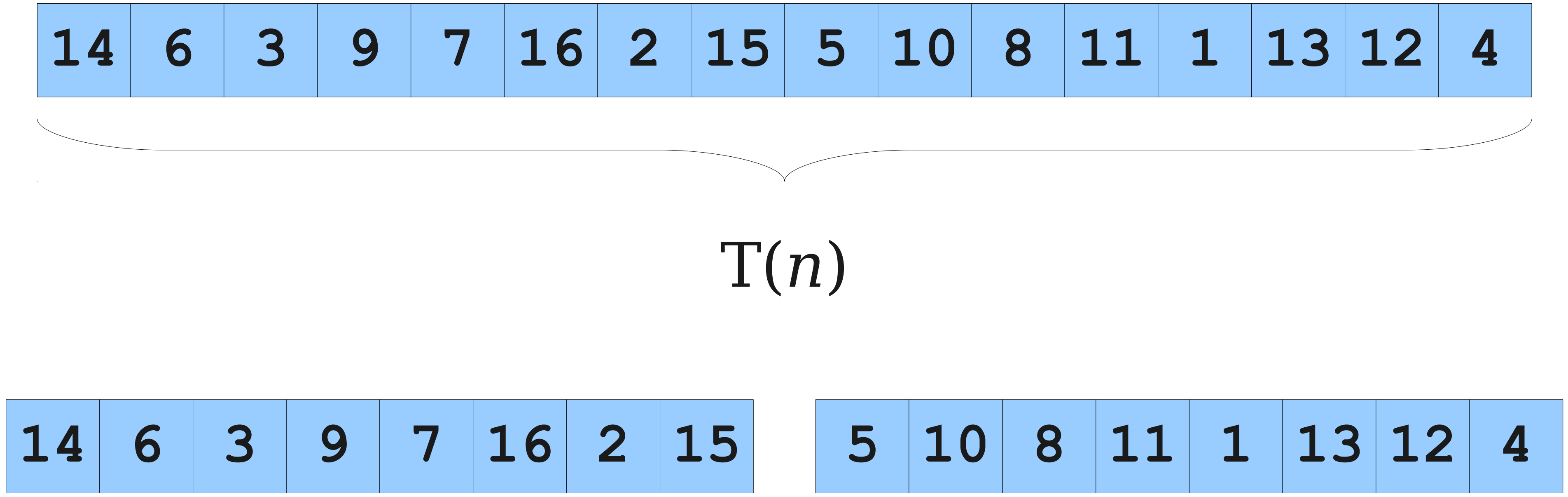
Thinking About O(n^2)

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

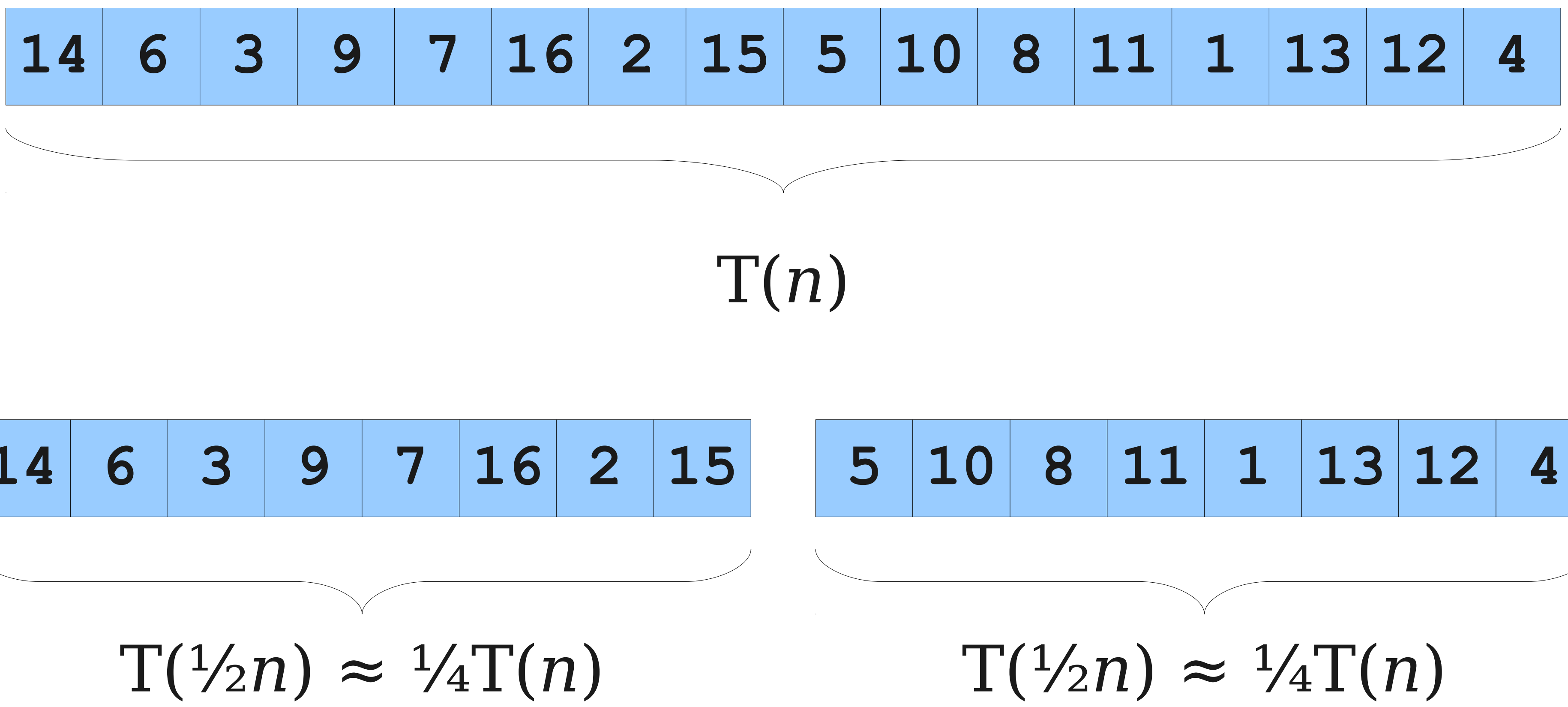
Thinking About O(n^2)



Thinking About O(n^2)



Thinking About $O(n^2)$


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
$T(n)$
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$
$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

Thinking About O(n^2)

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

$\underbrace{\hphantom{xxxxxxxxxxxxxxxxxxxxxxxx}}_{}$

$T(n)$

2	3	6	7	9	14	15	16

$\underbrace{\hphantom{xxxxxxxxxxxx}}_{}$

$T(\tfrac{1}{2}n) \approx \tfrac{1}{4}T(n)$

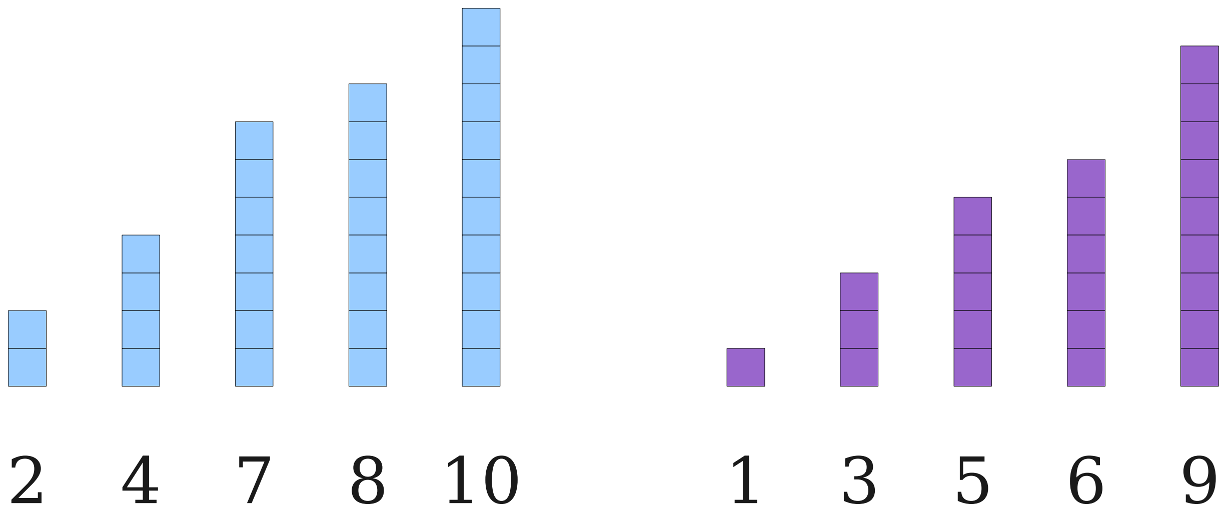
1	4	5	8	10	11	12	13

$\underbrace{\hphantom{xxxxxxxxxxxx}}_{}$

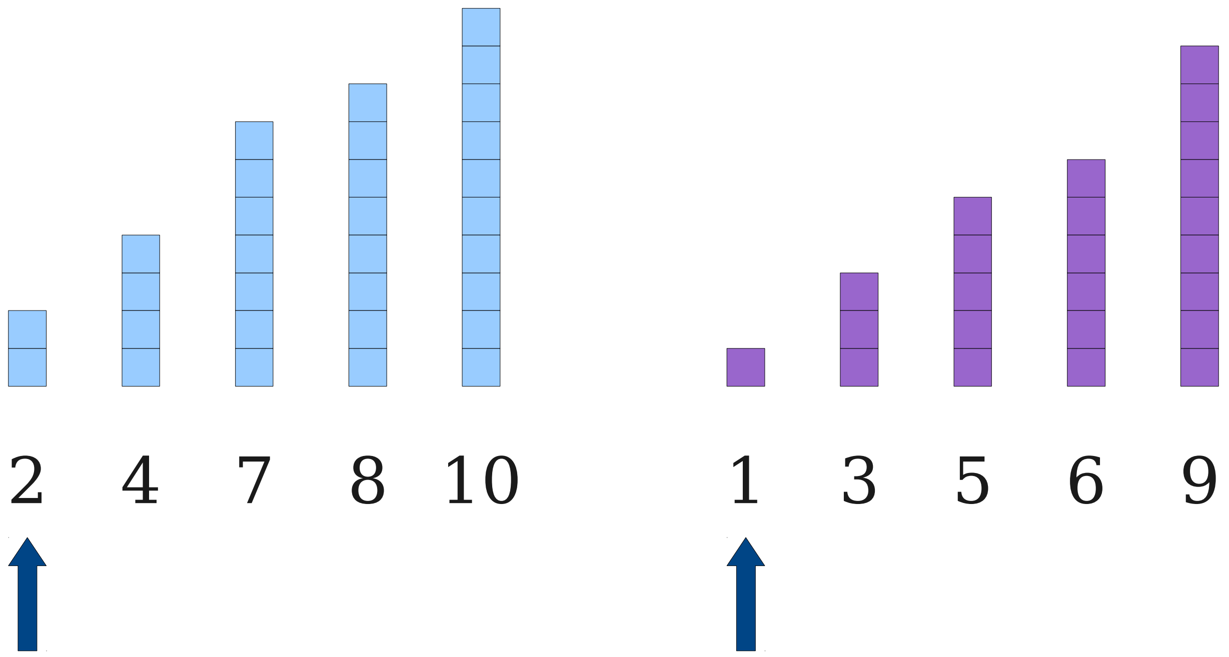
$T(\tfrac{1}{2}n) \approx \tfrac{1}{4}T(n)$

The Key Insight: **Merge**

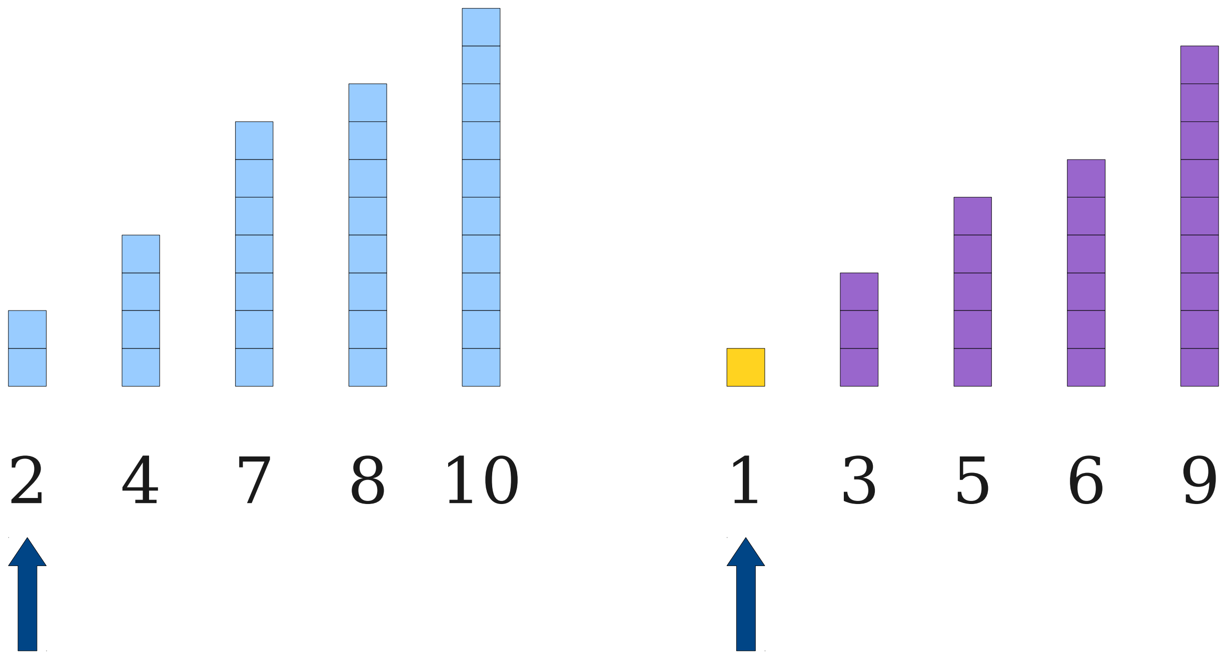
The Key Insight: **Merge**



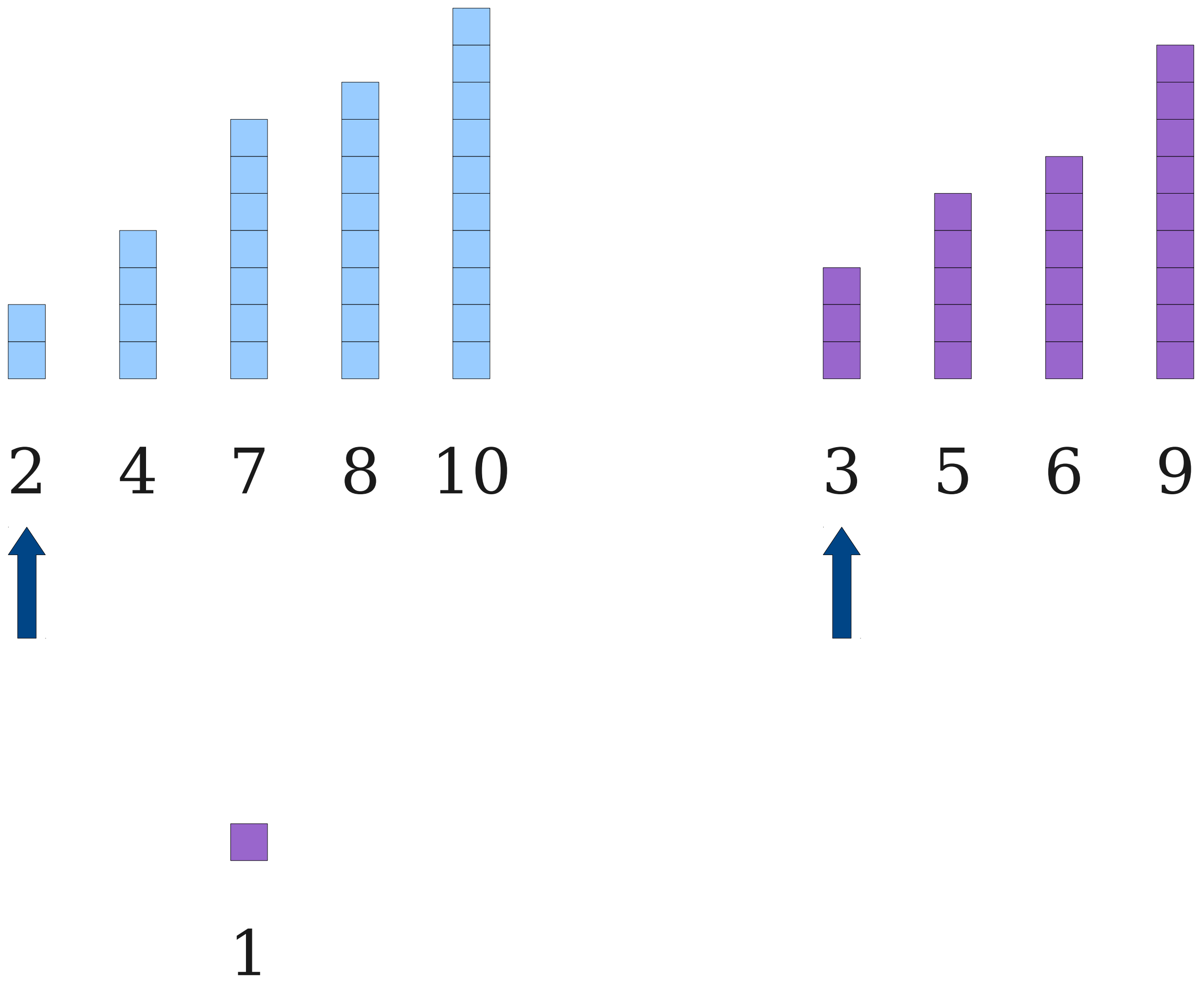
The Key Insight: **Merge**



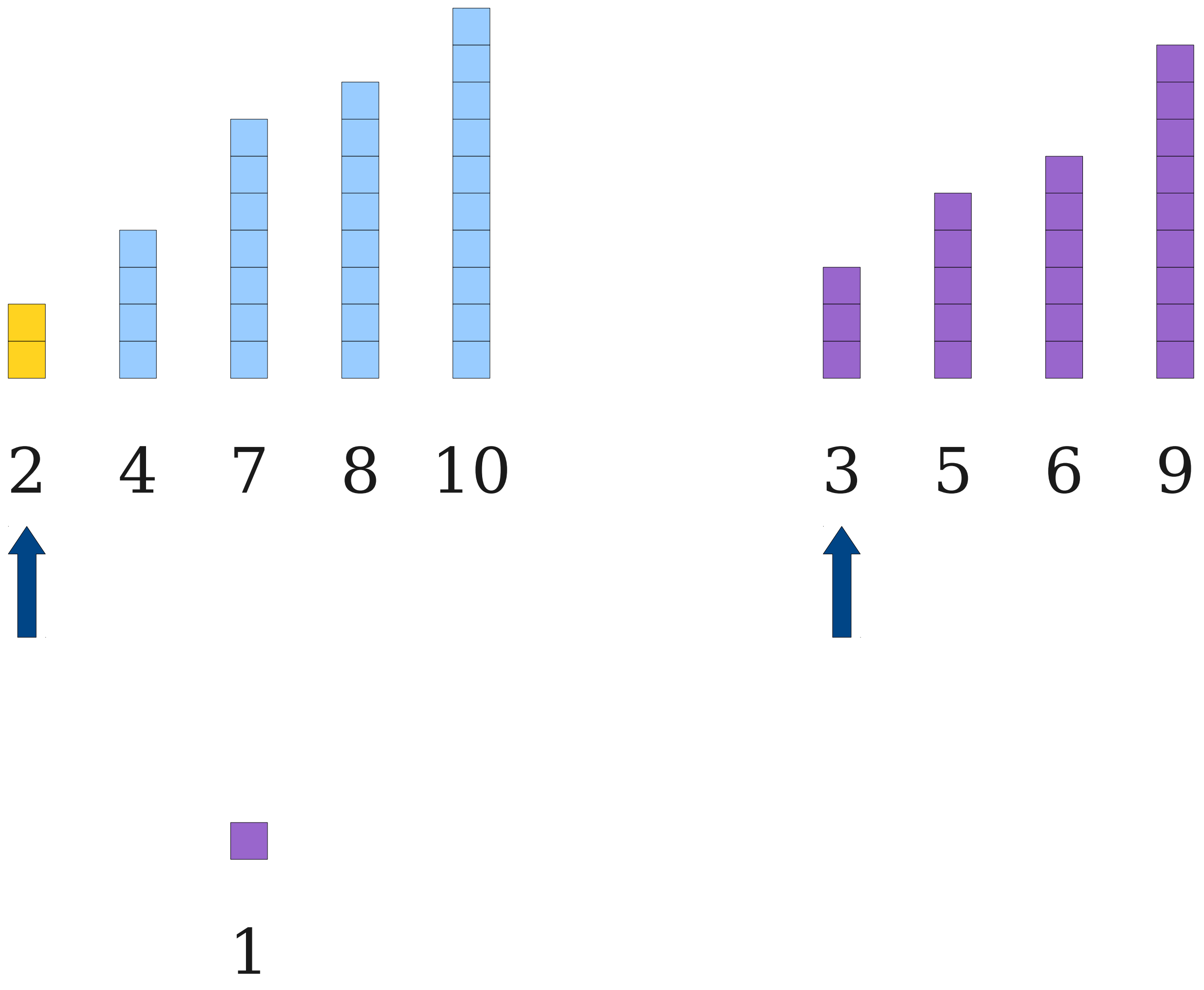
The Key Insight: **Merge**



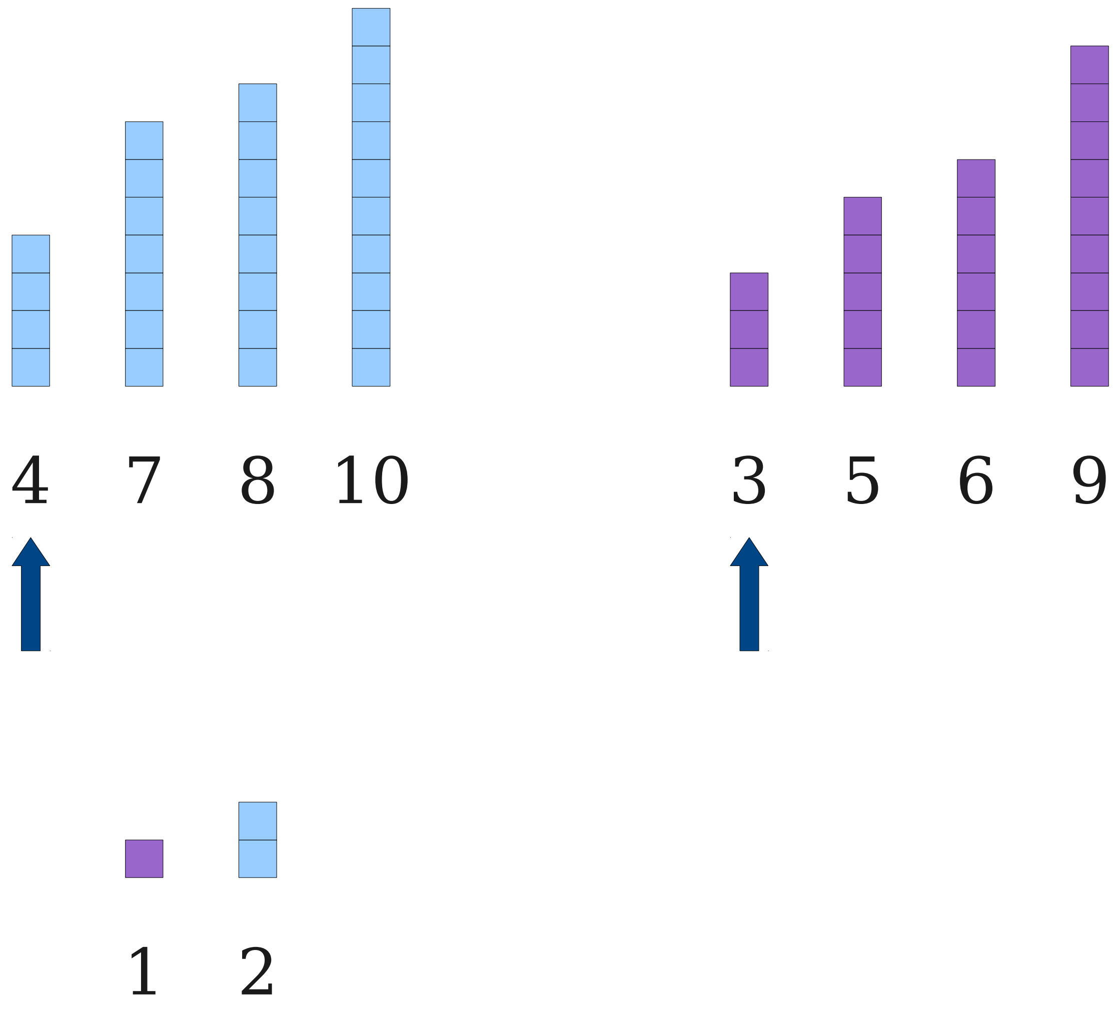
The Key Insight: **Merge**



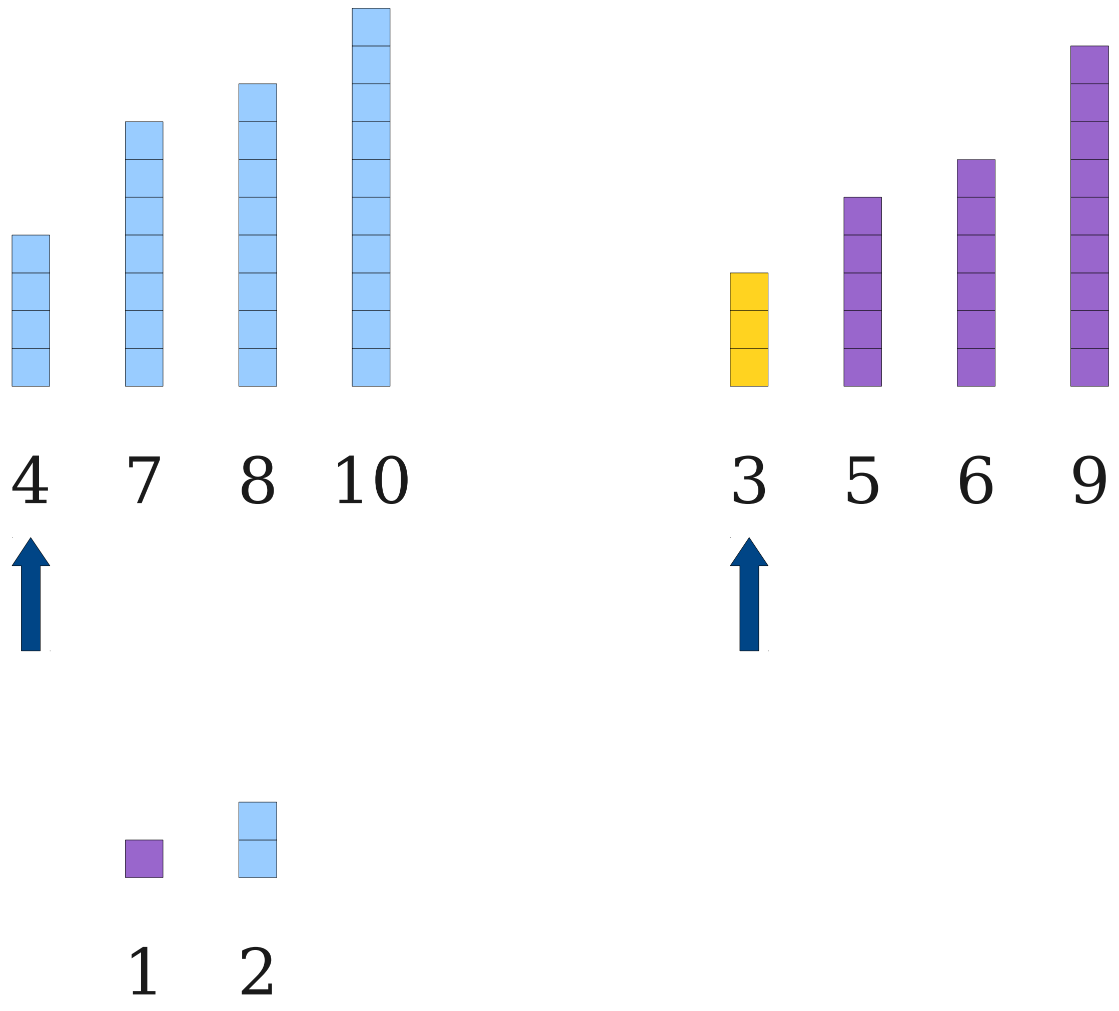
The Key Insight: **Merge**



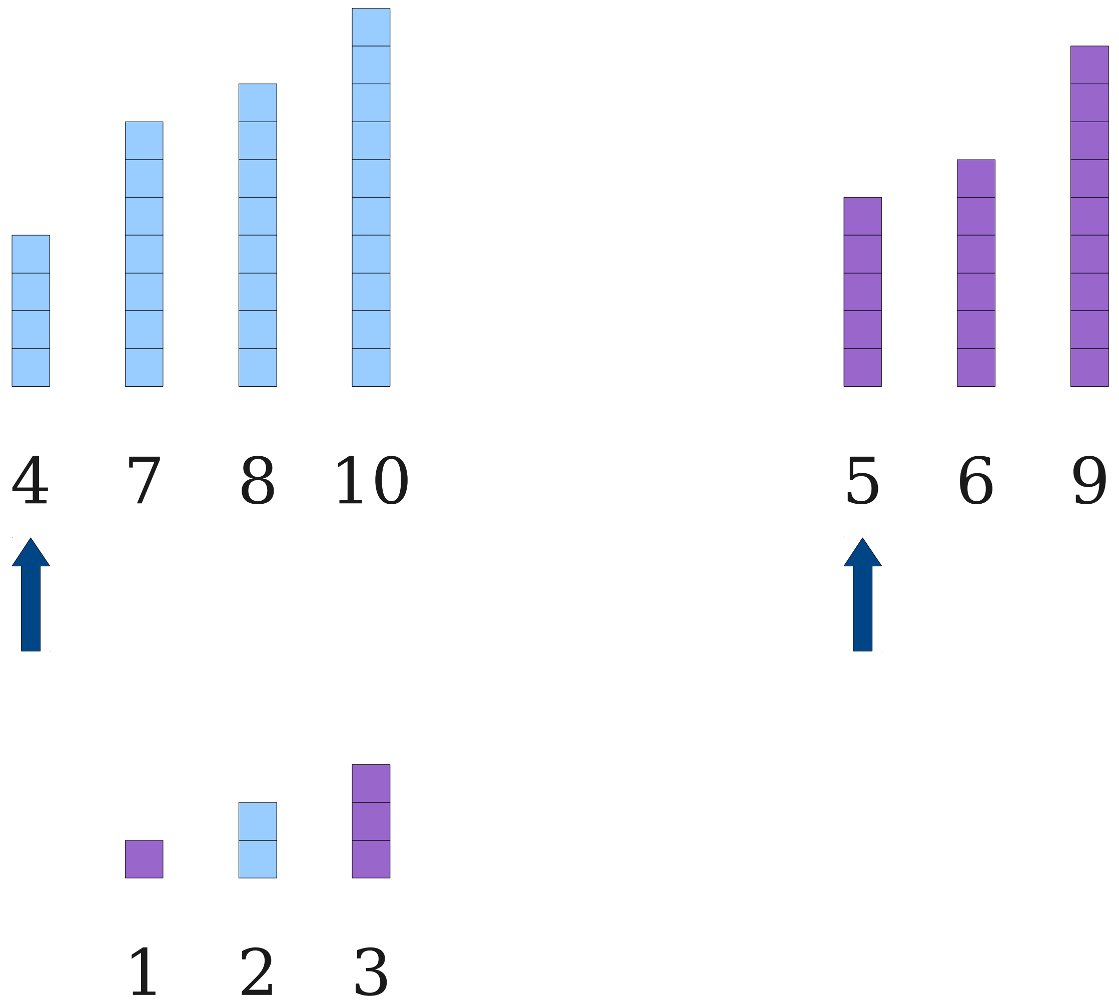
The Key Insight: **Merge**



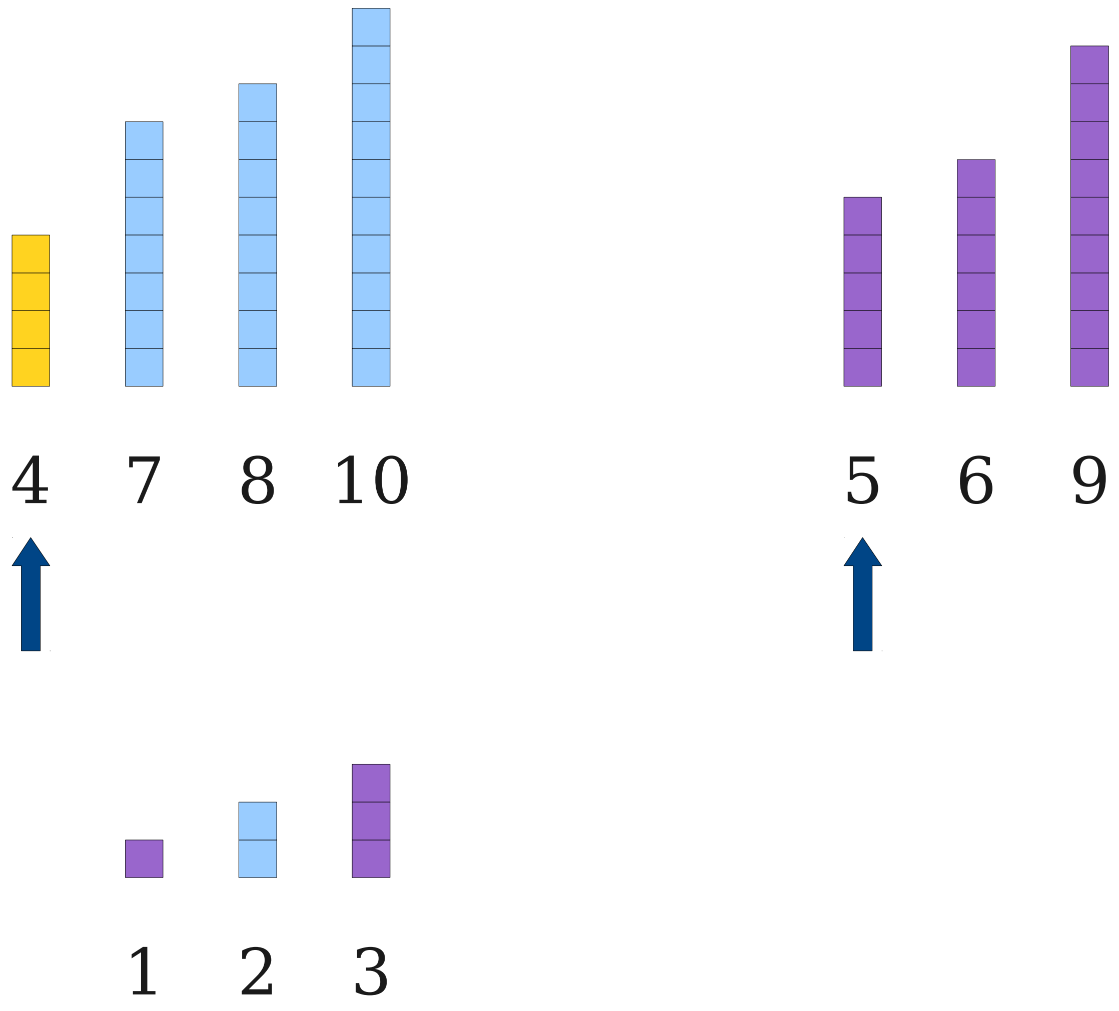
The Key Insight: **Merge**



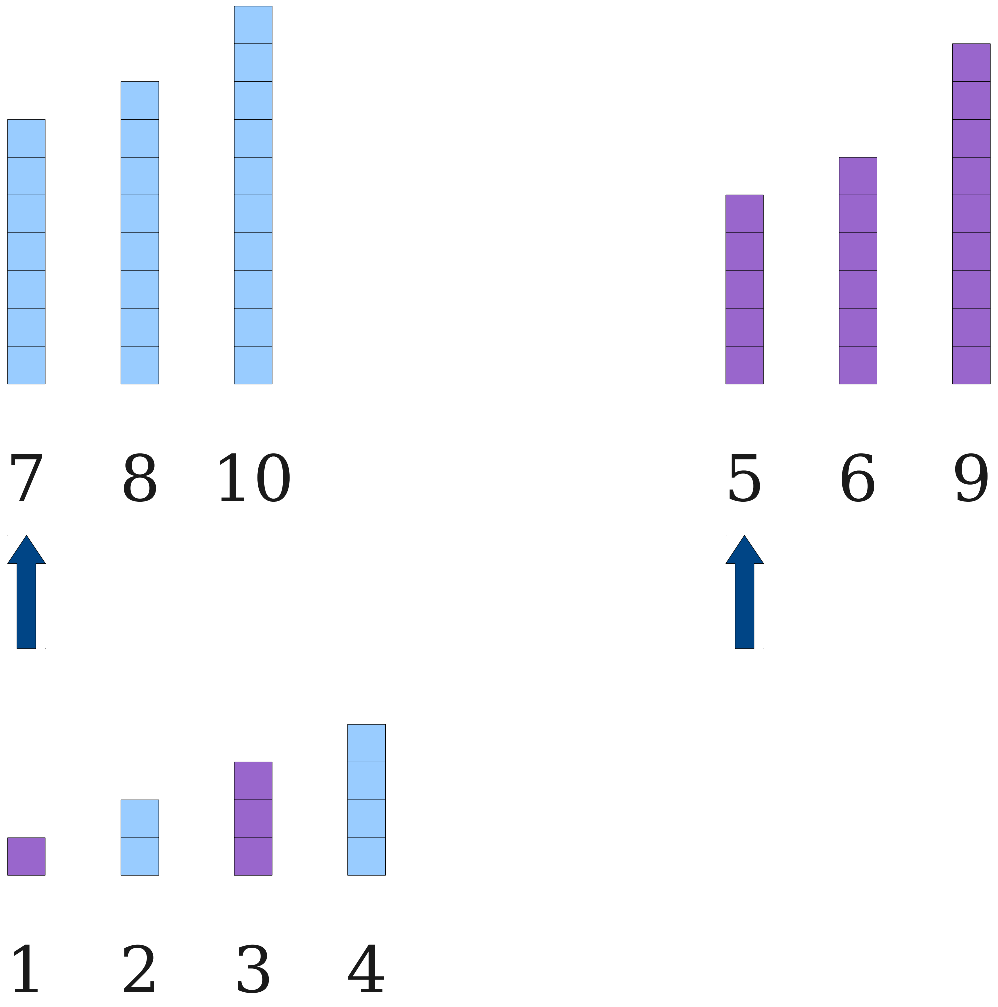
The Key Insight: **Merge**



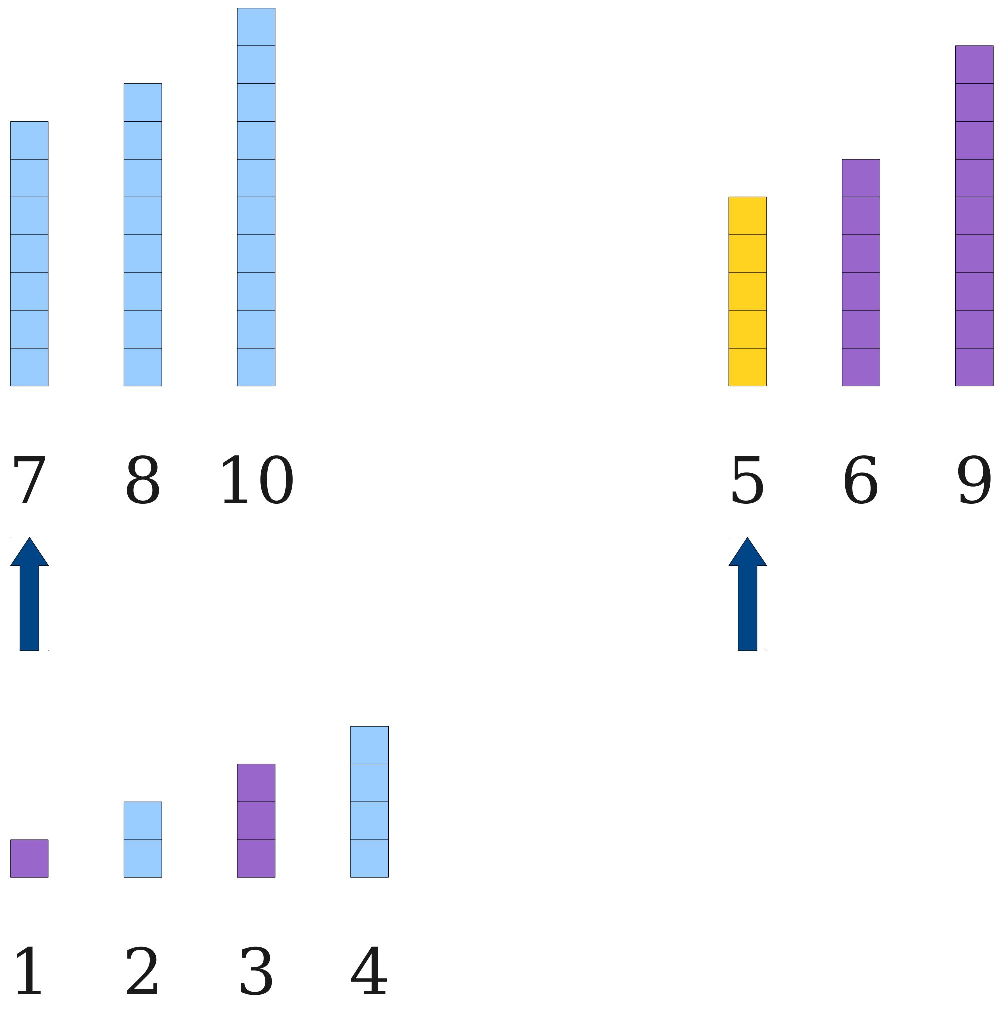
The Key Insight: **Merge**



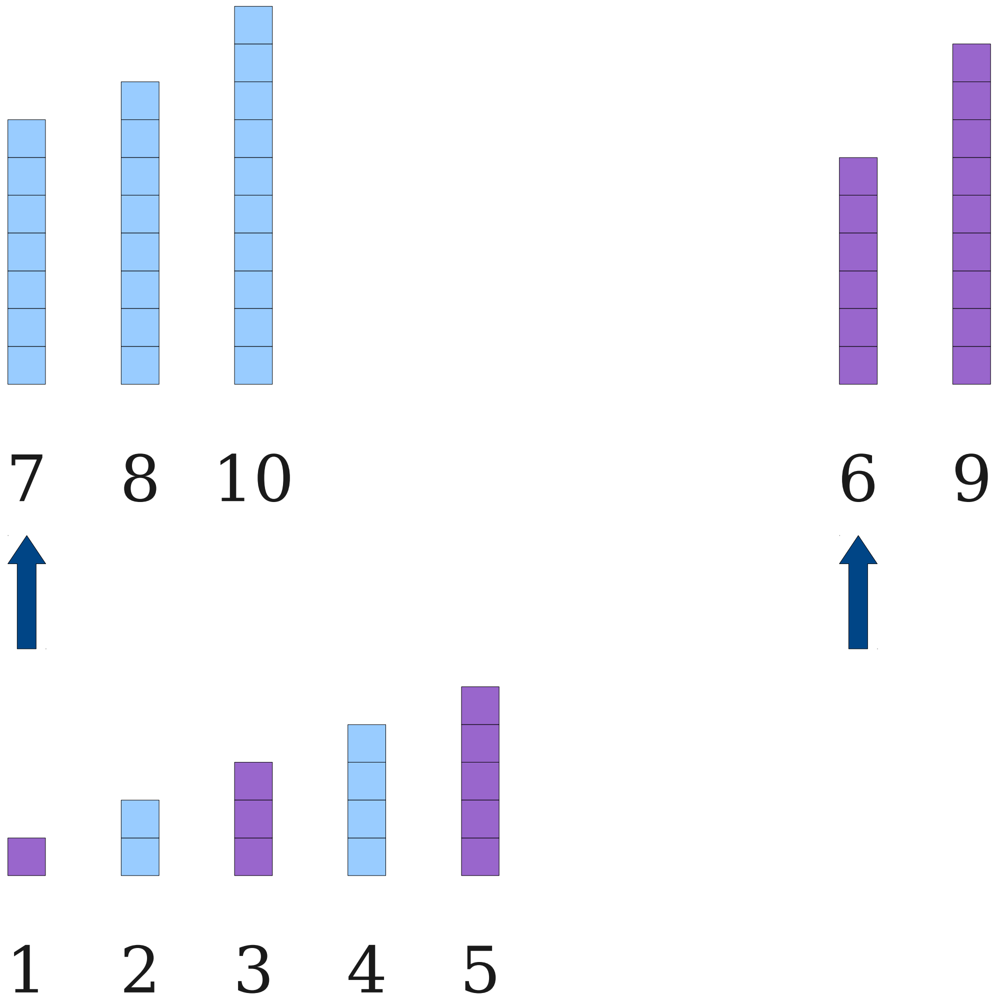
The Key Insight: **Merge**



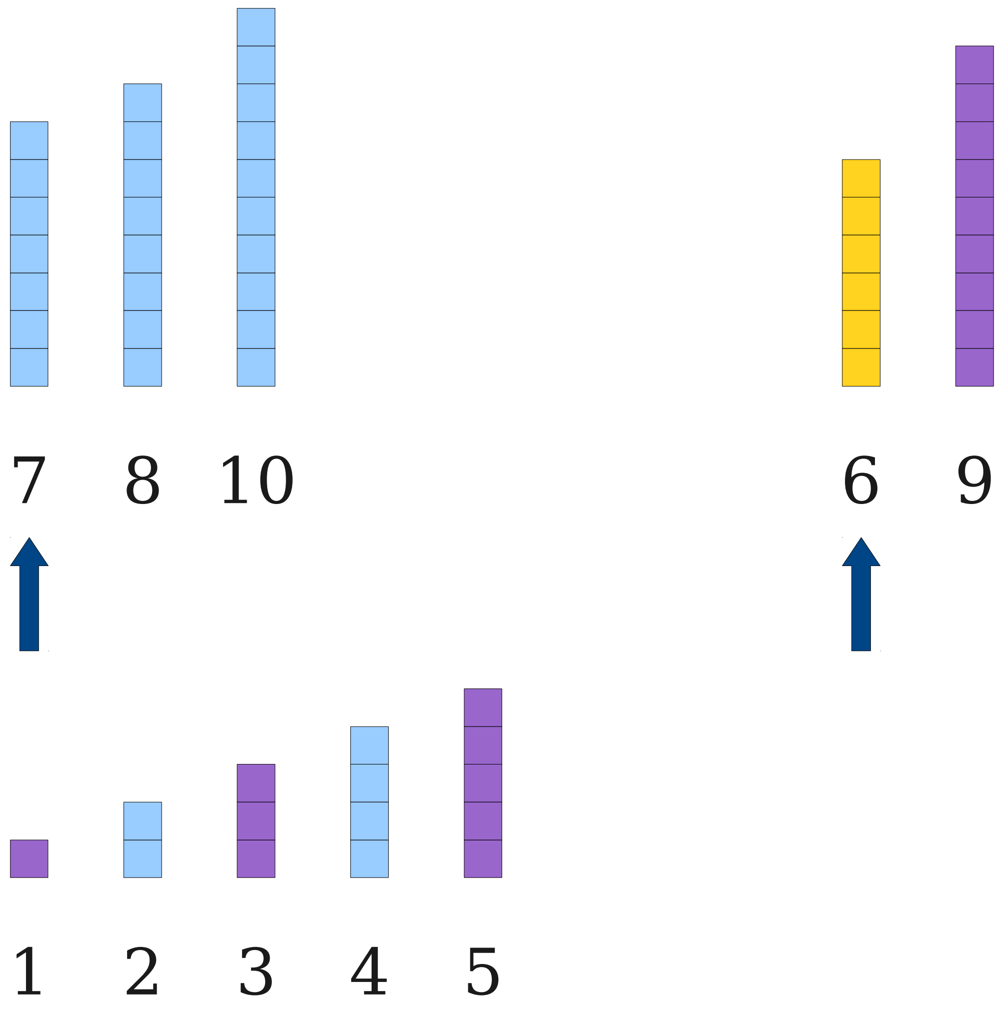
The Key Insight: **Merge**



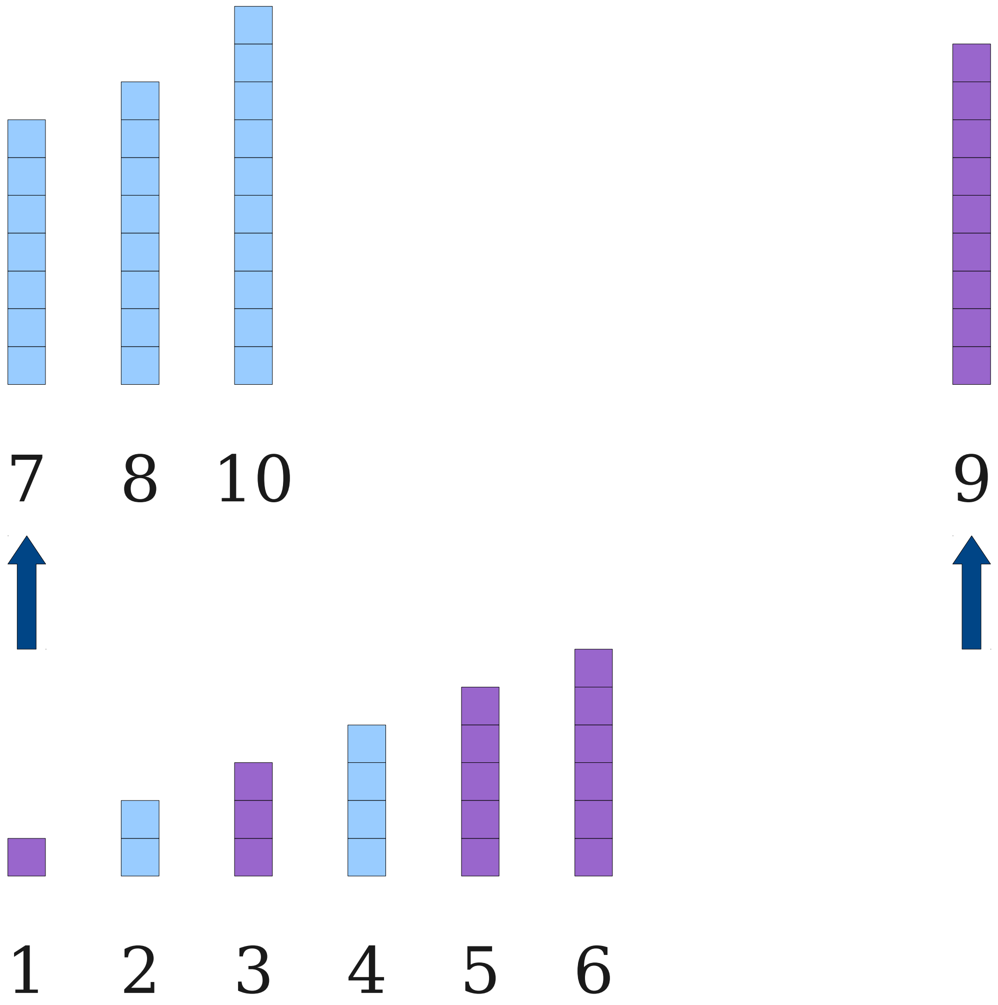
The Key Insight: **Merge**



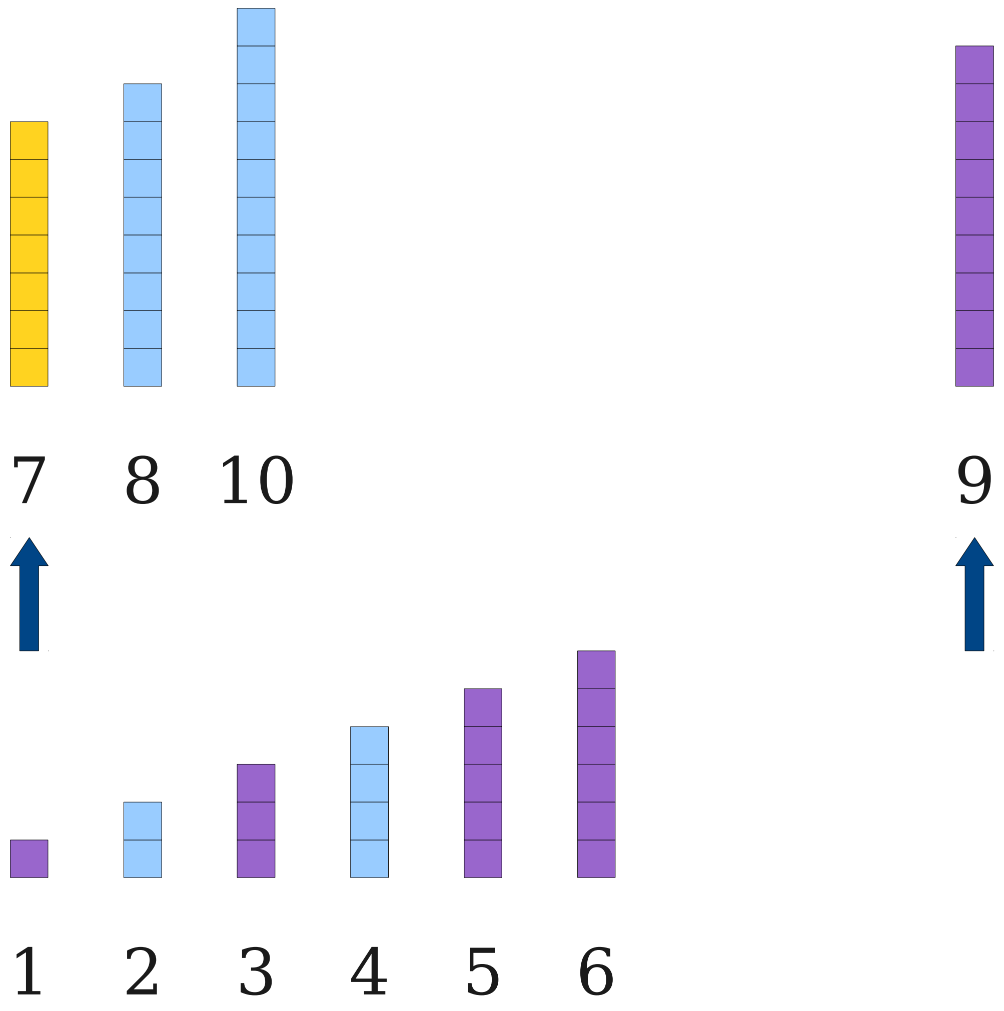
The Key Insight: **Merge**



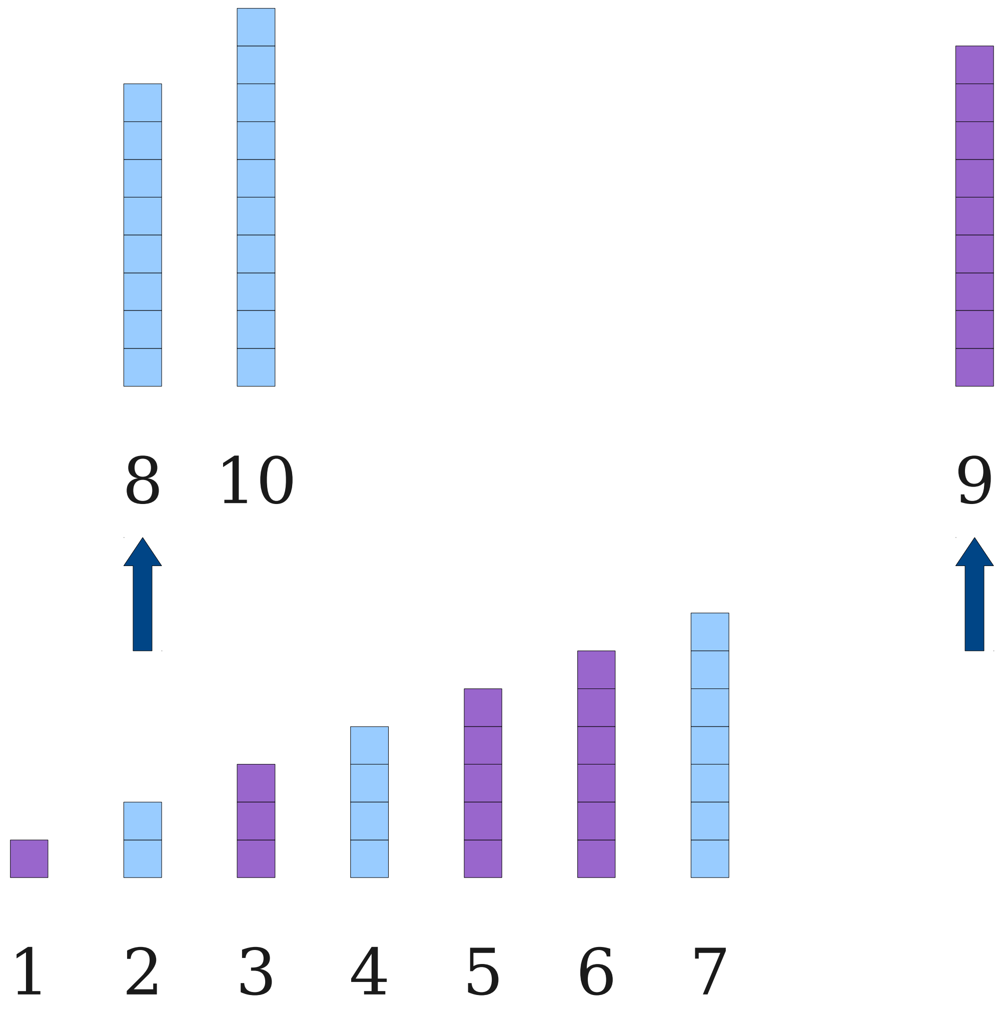
The Key Insight: **Merge**



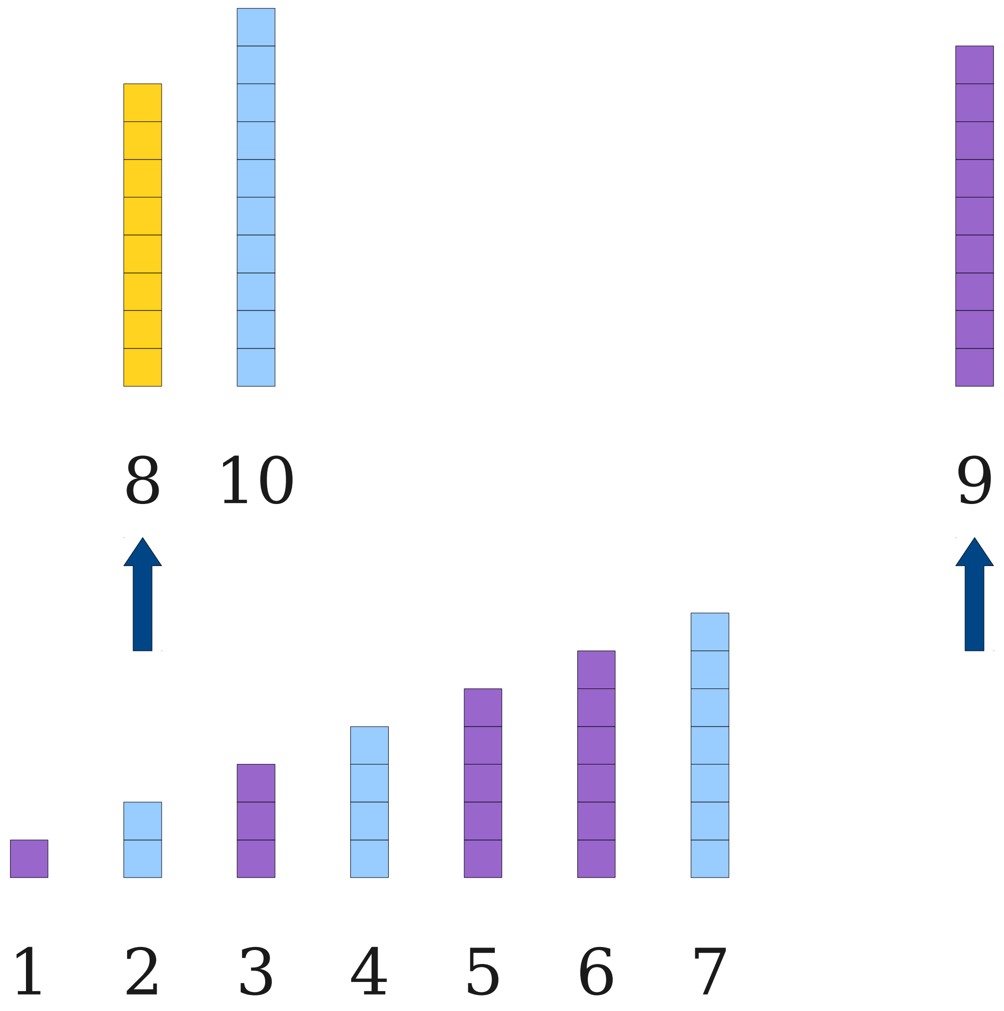
The Key Insight: **Merge**



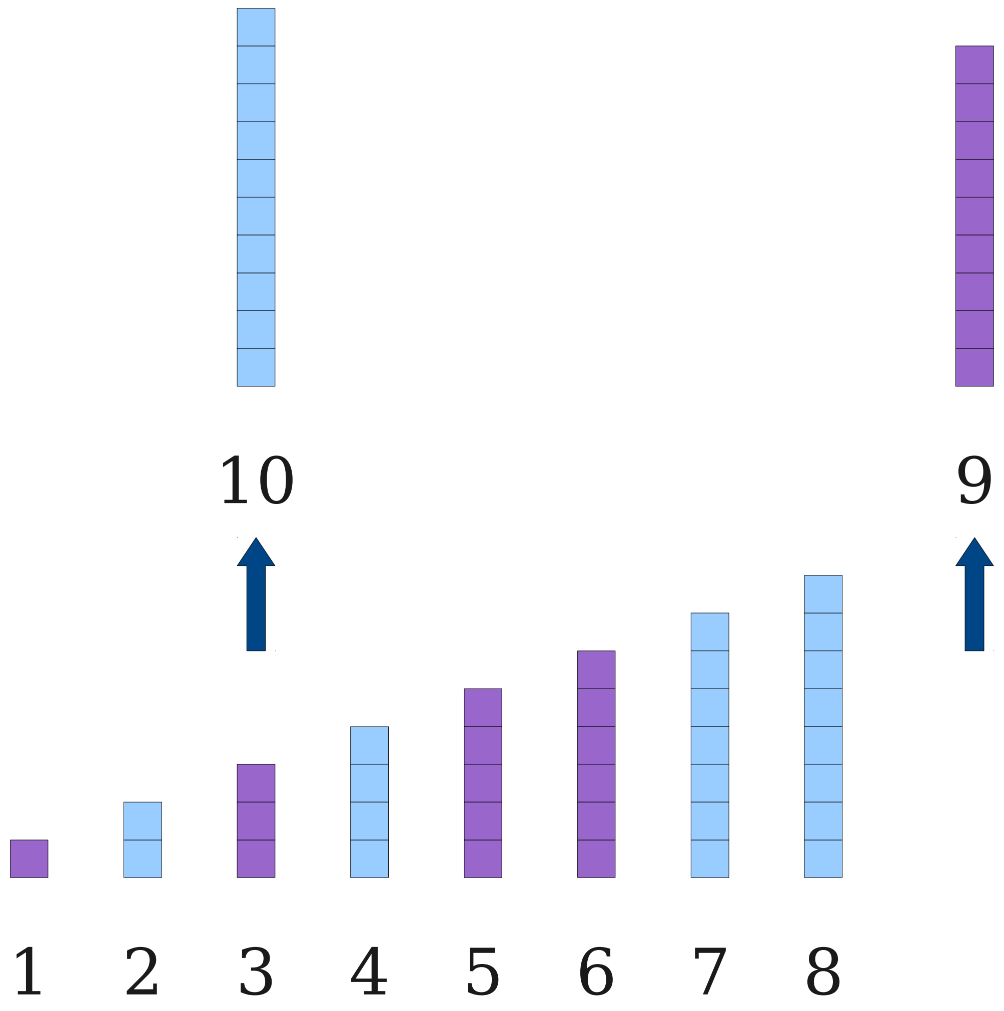
The Key Insight: **Merge**



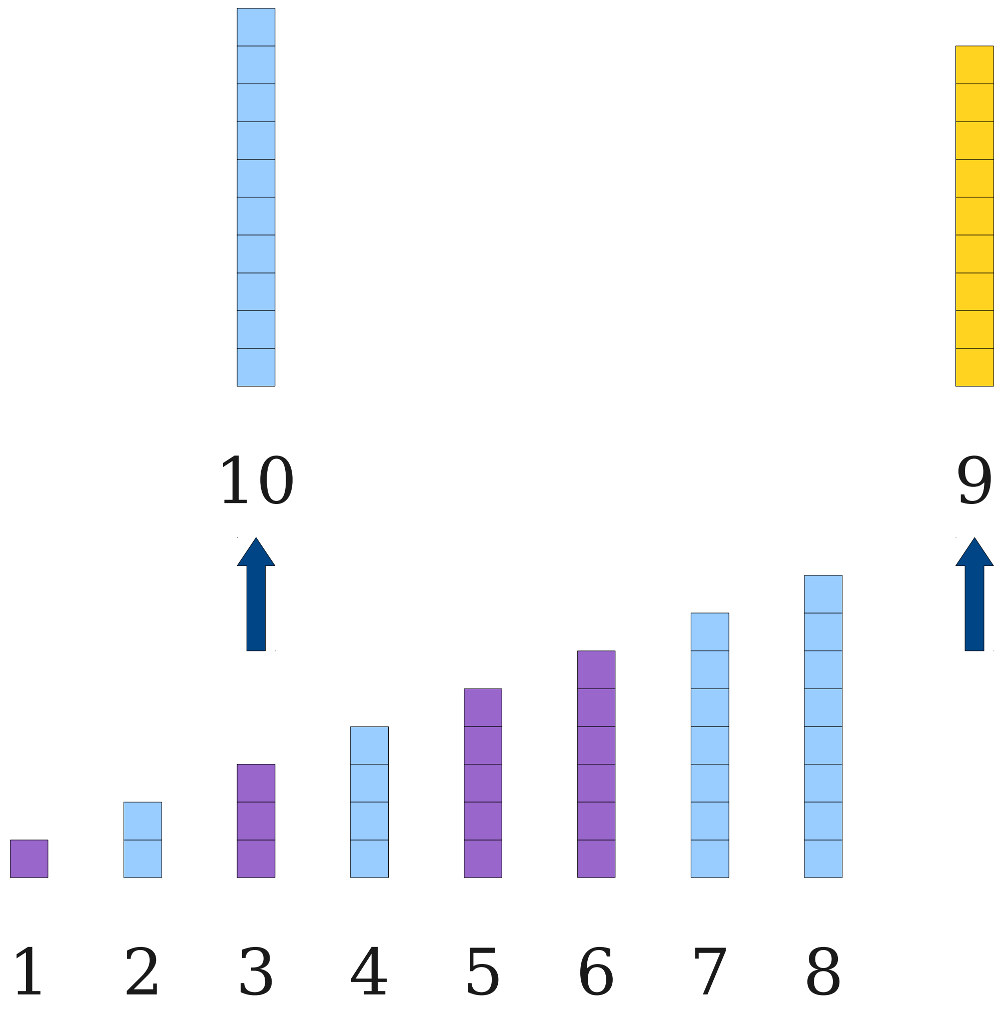
The Key Insight: **Merge**



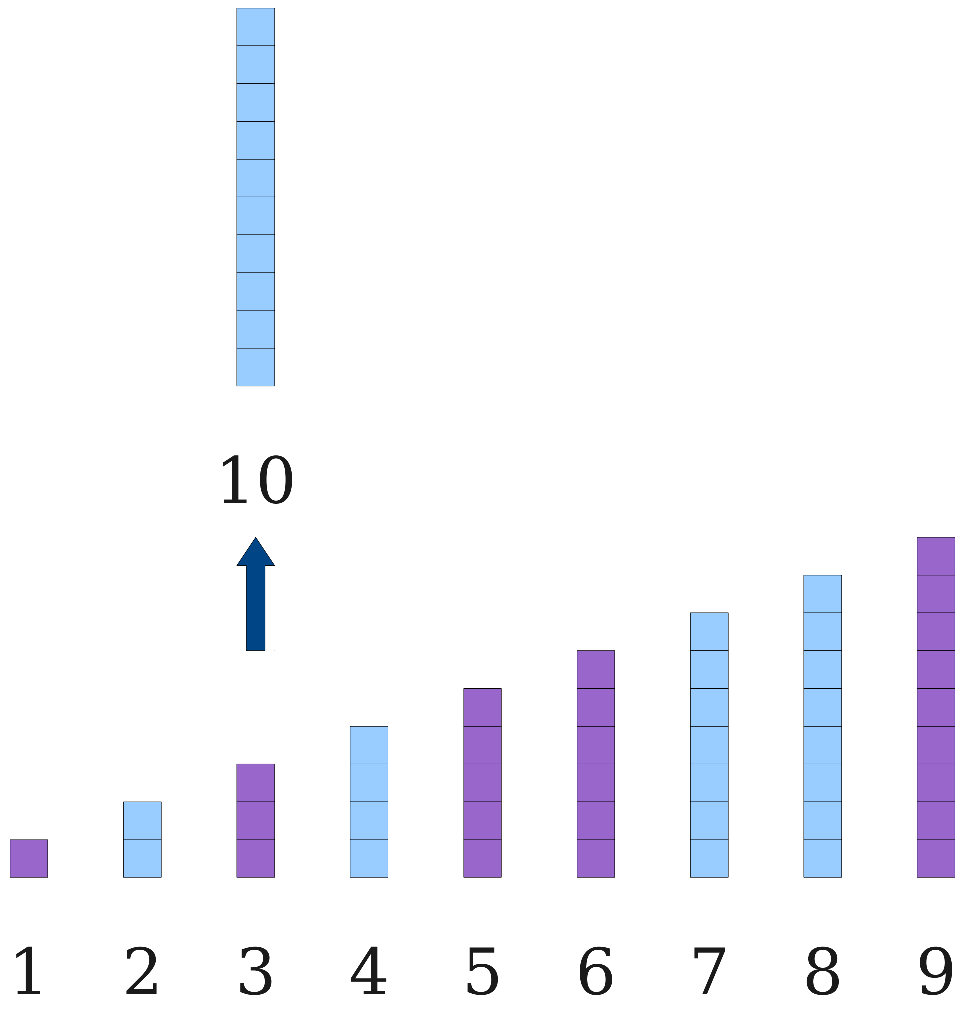
The Key Insight: **Merge**



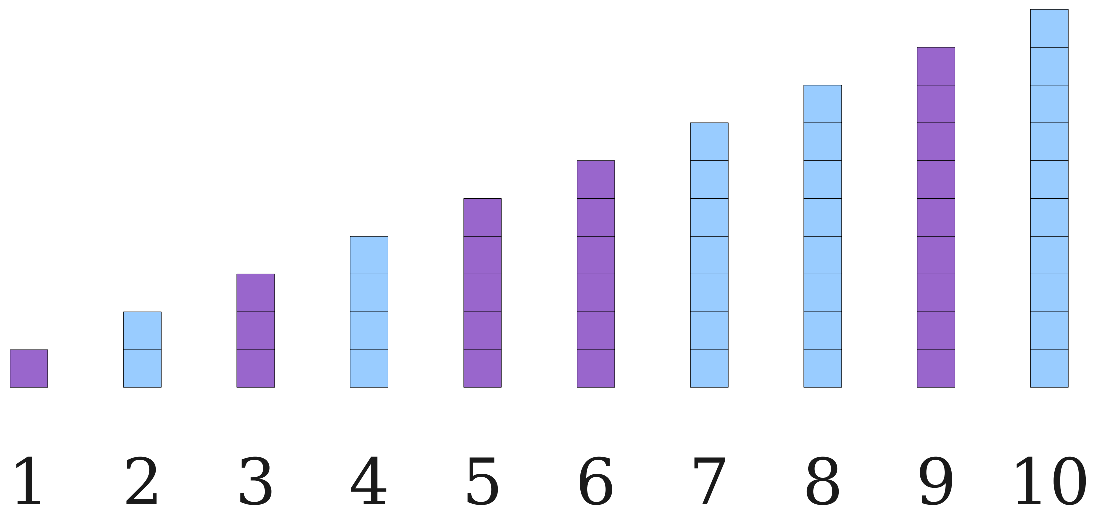
The Key Insight: **Merge**



The Key Insight: **Merge**



The Key Insight: **Merge**

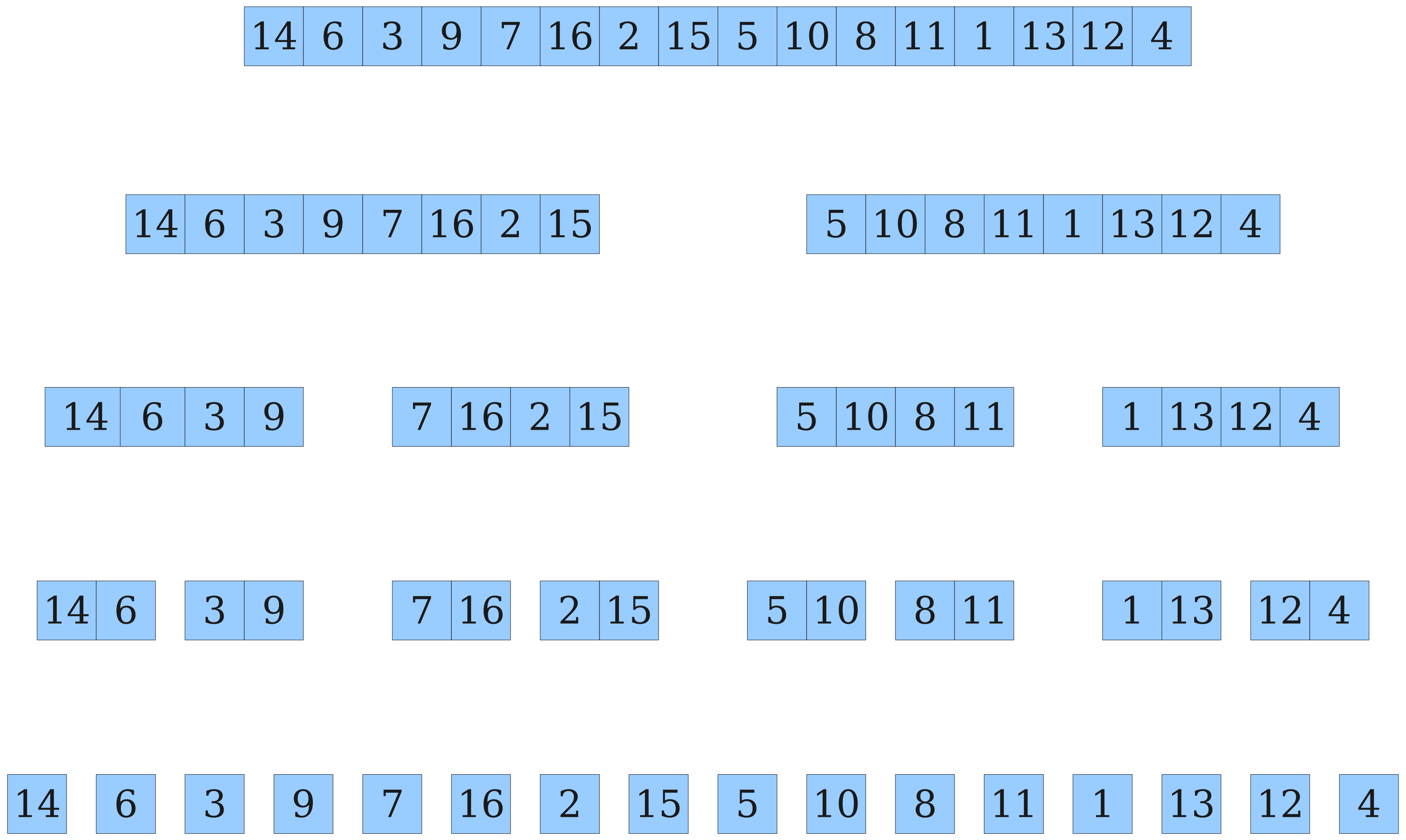


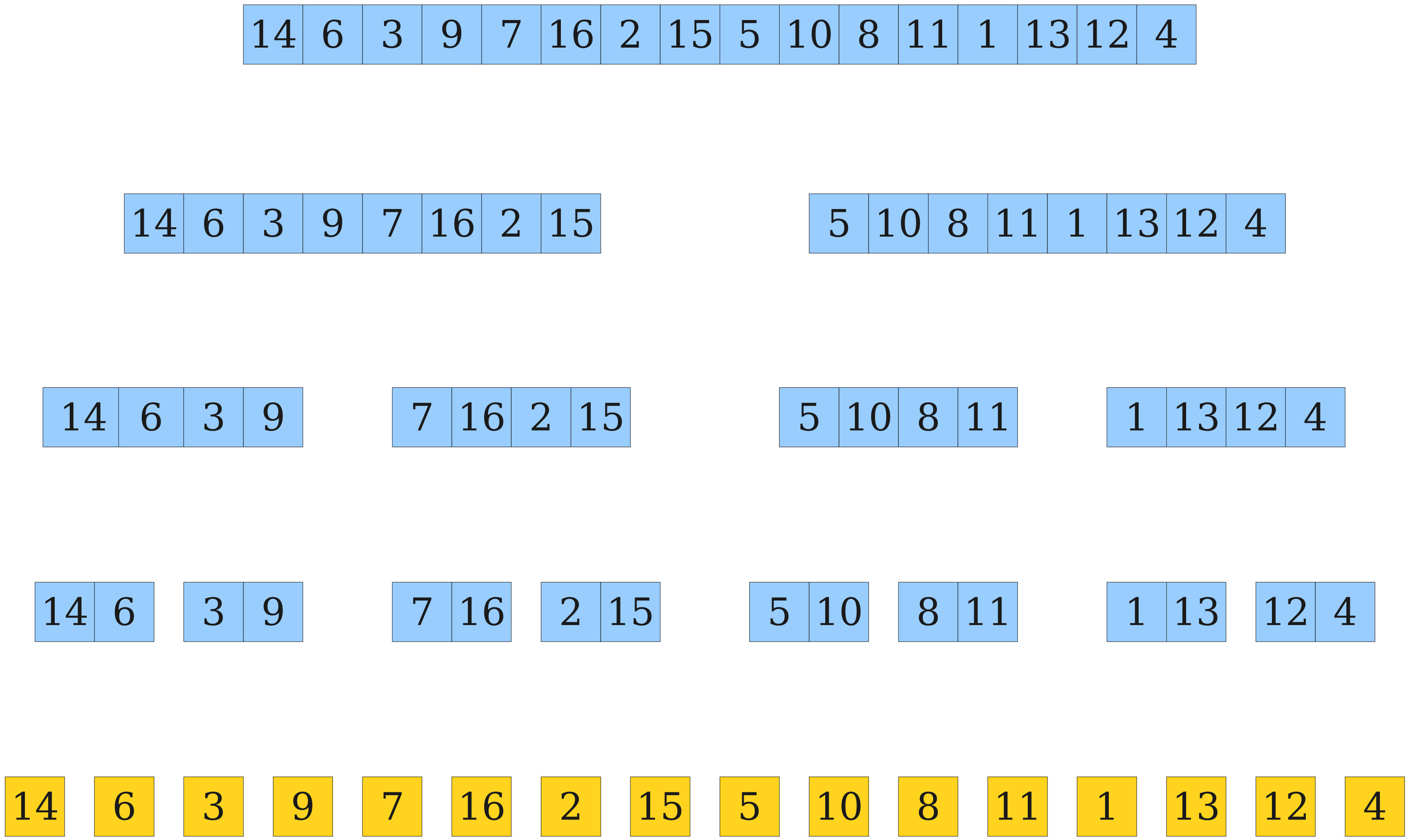
```
procedure merge(list A, list B):
   let result be an empty list.
   while both A and B are nonempty:
      if head(A) < head(B):
         append head(A) to result
         remove head(A) from A
      else:
         append head(B) to result
         remove head(B) from B

   append all elements remaining in A to result
   append all elements remaining in B to result

   return result
```

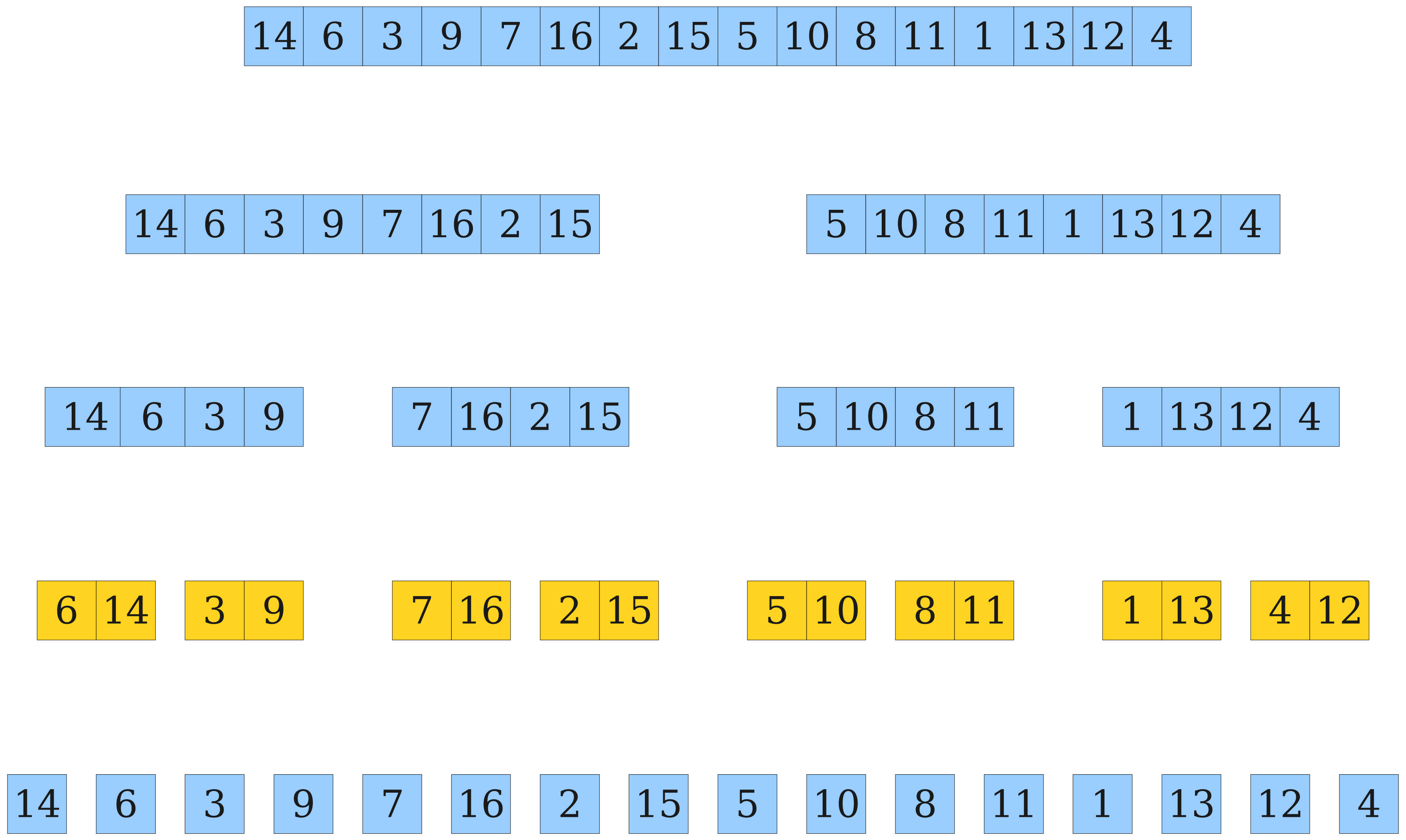
Complexity: $\boldsymbol{\Theta(m + n)}$,
where m and n are the lengths of the input lists.

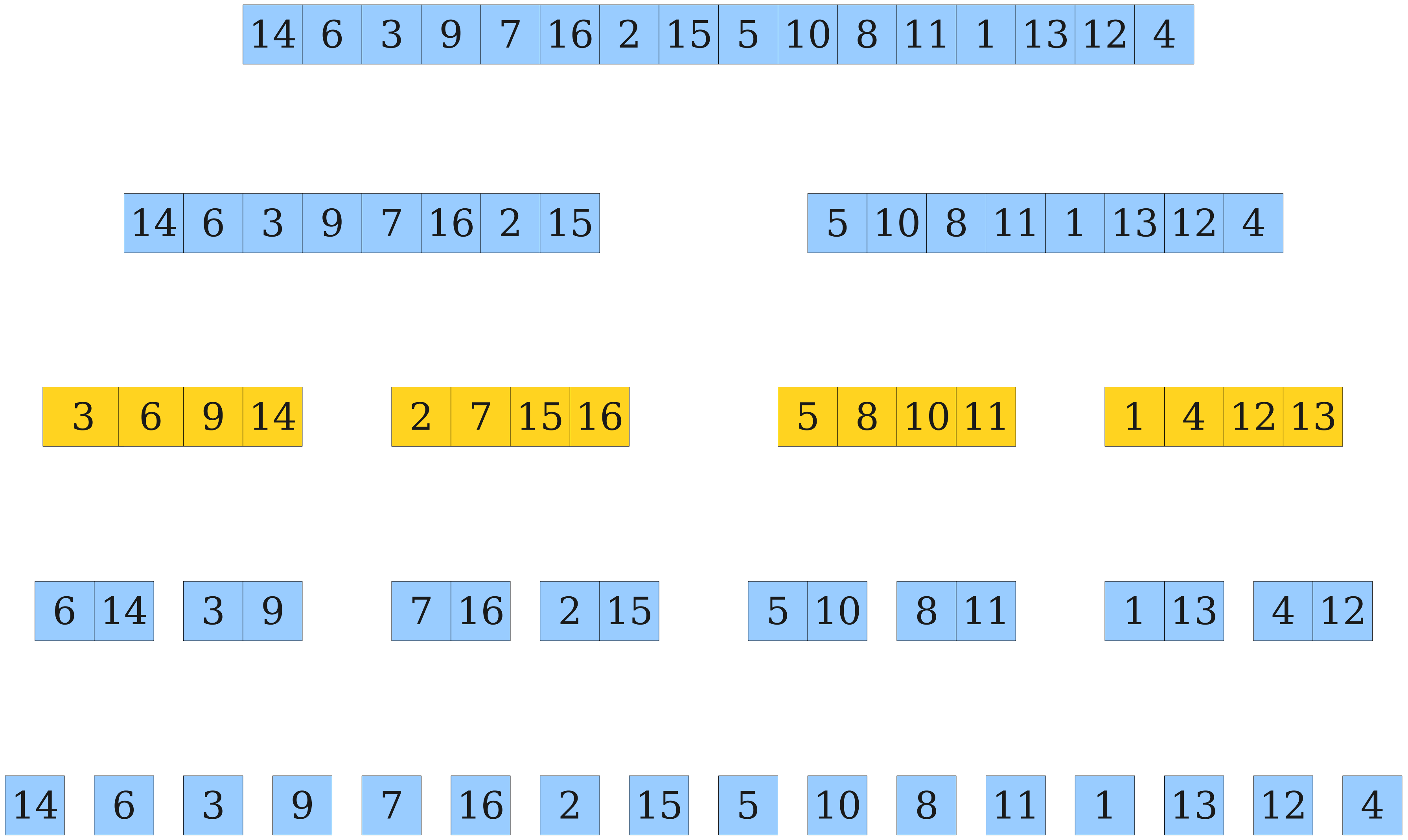
Motivating Mergesort

- Splitting the input array in half, sorting each half, and merging them back together will take roughly half as long as soring the original array.
- So why not split the array into fourths? Or eighths?
- **Question**: What happens if we *never stop splitting?*

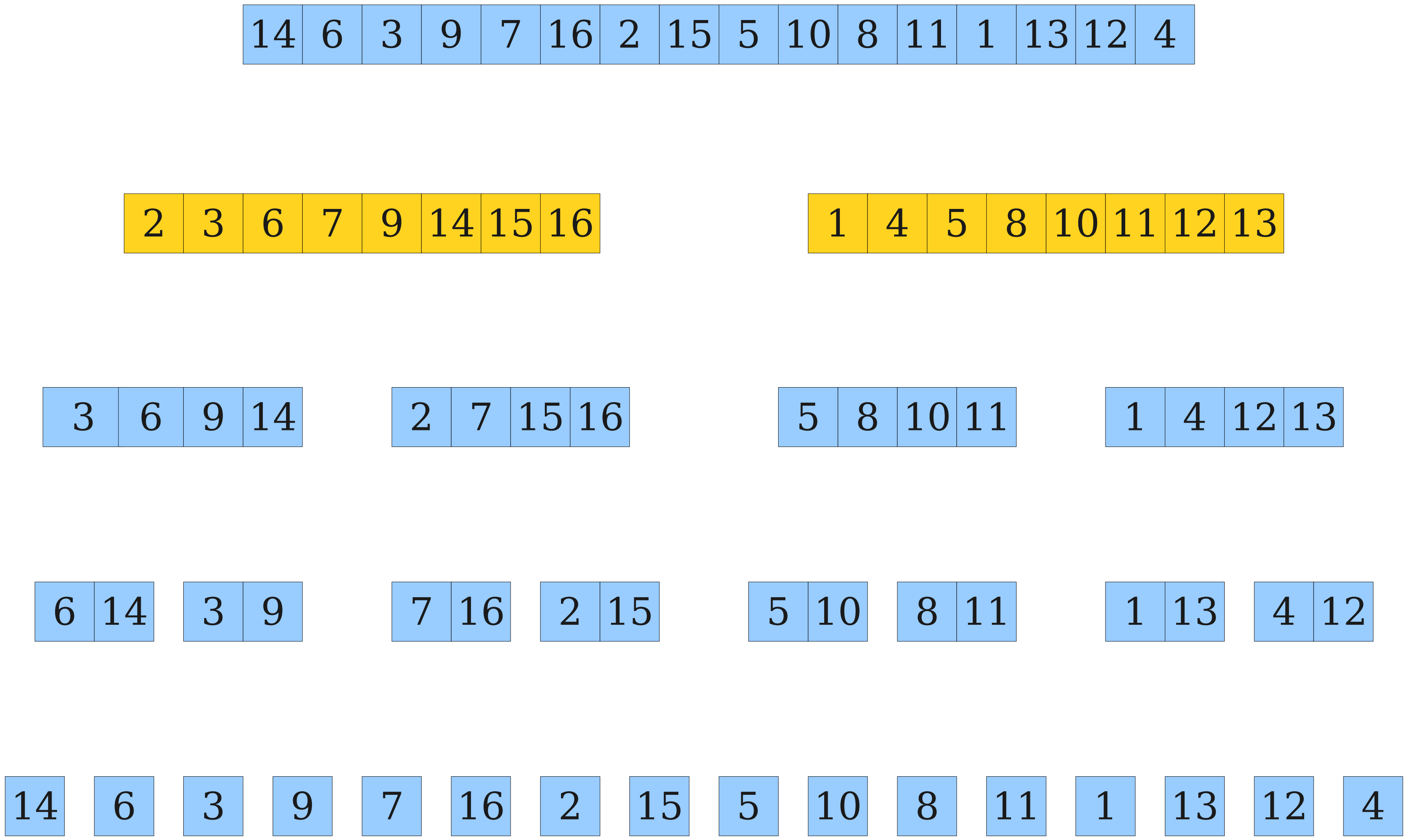

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

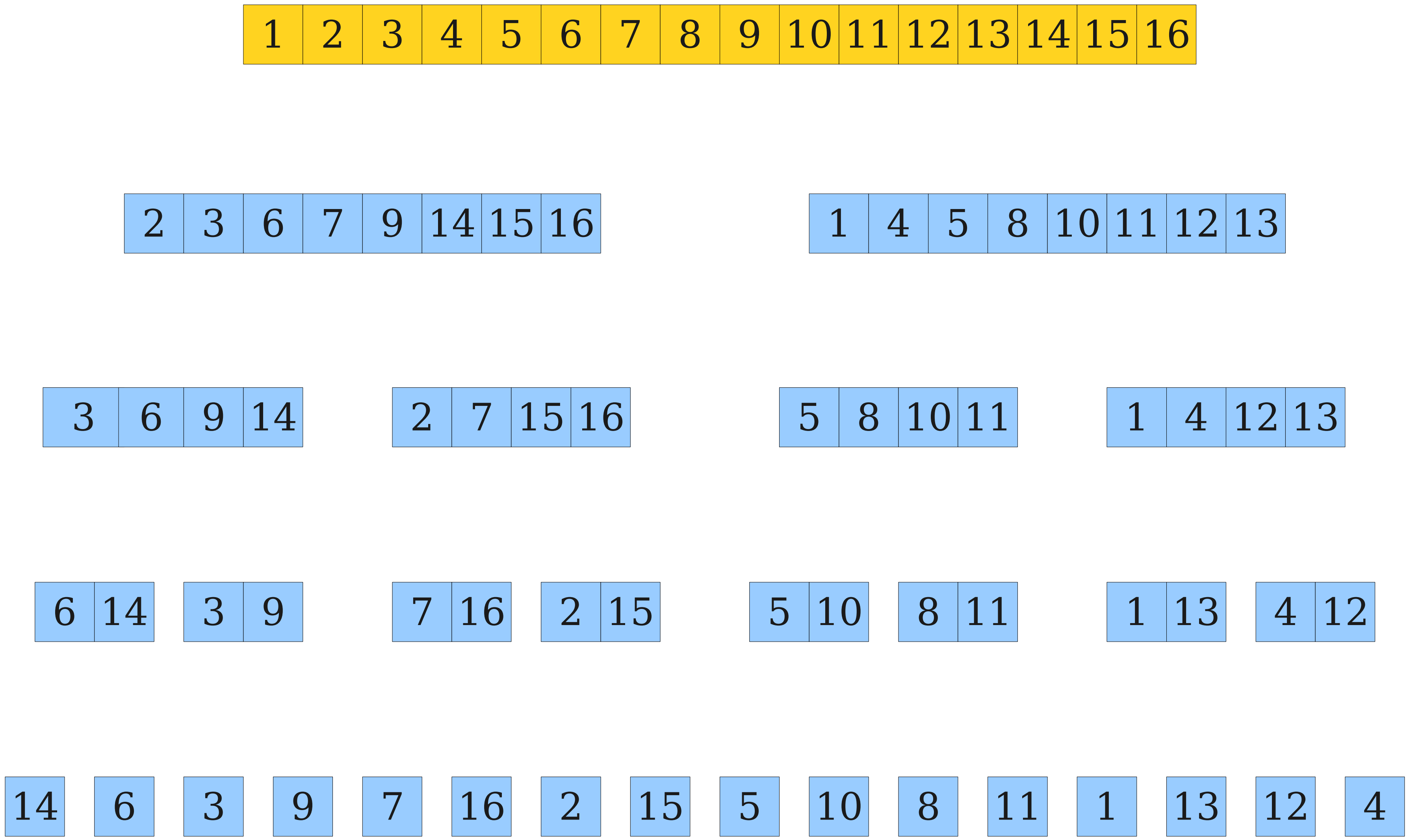

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4



14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4

High-Level Idea

- A recursive sorting algorithm!
- **Base Case**:
  - An empty or single-element list is already sorted.
- **Recursive step**:
  - Break the list in half and recursively sort each part.
  - Merge the sorted halves back together.
- This algorithm is called *mergesort*.

```
procedure mergesort(list A):
   if length(A) ≤ 1:
      return A

   let left  be the first  half of the elements of A
   let right be the second half of the elements of A

   return merge(mergesort(left), mergesort(right))
```

What is the complexity of mergesort?

```
procedure mergesort(list A):
   if length(A) ≤ 1:
      return A

   let left  be the first  half of the elements of A
   let right be the second half of the elements of A

   return merge(mergesort(left), mergesort(right))
```

```
procedure mergesort(list A):
   if length(A) ≤ 1:
      return A

   let left  be the first  half of the elements of A
   let right be the second half of the elements of A

   return merge(mergesort(left), mergesort(right))
```

$$T(0) = \Theta(1)$$

```
procedure mergesort(list A):
   if length(A) ≤ 1:
      return A

   let left  be the first  half of the elements of A
   let right be the second half of the elements of A

   return merge(mergesort(left), mergesort(right))
```

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$

```
procedure mergesort(list A):
   if length(A) ≤ 1:
      return A

   let left  be the first  half of the elements of A
   let right be the second half of the elements of A

   return merge(mergesort(left), mergesort(right))
```

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n / 2 \rceil) + T(\lfloor n / 2 \rfloor) + \Theta(n)$$

Recurrence Relations

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier values.
- In our case, we get this recurrence for the runtime of mergesort:

$$\boxed{\begin{array}{l} T(0) = \Theta(1) \\ T(1) = \Theta(1) \\ T(n) = T(\lceil n / 2 \rceil) + T(\lfloor n / 2 \rfloor) + \Theta(n) \end{array}}$$

- We can **solve** a recurrence by finding an explicit expression for its terms, or by finding an asymptotic bound on its growth rate.
- How do we solve this recurrence?

Simplifying our Recurrence

- It is often difficult to solve recurrences involving floors and ceilings, as ours does.

$$\boxed{\begin{aligned}&T(0) = \Theta(1)\\&T(1) = \Theta(1)\\&T(n) = T(\lceil n / 2\rceil) + T(\lfloor n / 2\rfloor) + \Theta(n)\end{aligned}}$$

- Note that if we only consider $n = 1, 2, 4, 8, 16, \ldots$, then the floors and ceilings are always equivalent to standard division.
- **Simplifying Assumption 1:** We will only consider the recurrence as applied to powers of two.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- It is often difficult to solve recurrences involving floors and ceilings, as ours does.

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) = T(\lceil n / 2 \rceil) + T(\lfloor n / 2 \rfloor) + \Theta(n) \end{aligned}}$$

- Note that if we only consider $n = 1, 2, 4, 8, 16, \ldots$, then the floors and ceilings are always equivalent to standard division.
- **Simplifying Assumption 1:** We will only consider the recurrence as applied to powers of two.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- It is often difficult to solve recurrences involving floors and ceilings, as ours does.

$$\boxed{\begin{array}{l} T(1) = \Theta(1) \\ T(n) = T(n / 2) + T(n / 2) + \Theta(n) \end{array}}$$

- Note that if we only consider $n = 1, 2, 4, 8, 16, \ldots$, then the floors and ceilings are always equivalent to standard division.
- **Simplifying Assumption 1:** We will only consider the recurrence as applied to powers of two.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- It is often difficult to solve recurrences involving floors and ceilings, as ours does.

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) = 2T(n / 2) + \Theta(n) \end{aligned}}$$

- Note that if we only consider $n = 1, 2, 4, 8, 16, \ldots$, then the floors and ceilings are always equivalent to standard division.
- **Simplifying Assumption 1:** We will only consider the recurrence as applied to powers of two.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- Without knowing the actual functions hidden by the Θ notation, we cannot get an exact value for the terms in this recurrence.

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) = 2T(n / 2) + \Theta(n) \end{aligned}}$$

- If the Θ(1) just hides a constant and Θ(n) just hides a multiple of n, this would be a lot easier to manipulate!

- **Simplifying Assumption 2:** We will pretend that Θ(1) hides some constant and Θ(n) hides a multiple of n.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- Without knowing the actual functions hidden by the Θ notation, we cannot get an exact value for the terms in this recurrence.

$$\boxed{\begin{aligned} &T(1) = c_1 \\ &T(n) = 2T(n / 2) + c_2 n \end{aligned}}$$

- If the Θ(1) just hides a constant and Θ(n) just hides a multiple of n, this would be a lot easier to manipulate!
- **Simplifying Assumption 2:** We will pretend that Θ(1) hides some constant and Θ(n) hides a multiple of n.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- Working with two constants c_1 and c_2 is most accurate, but it makes the math a *lot* harder.

$$\boxed{\begin{aligned} &T(1) = c_1 \\ &T(n) = 2T(n / 2) + c_2 n \end{aligned}}$$

- If all we care about is getting an asymptotic bound, these constants are unlikely to make a noticeable difference.
- **Simplifying Assumption 3:** Set $c = \max\{c_1, c_2\}$ and replace the equality with an upper bound.
- We need to justify why this is safe, which we'll do later.

Simplifying our Recurrence

- Working with two constants c_1 and c_2 is most accurate, but it makes the math a *lot* harder.

$$\boxed{\begin{aligned} &T(1) \leq c \\ &T(n) \leq 2T(n / 2) + cn \end{aligned}}$$

- If all we care about is getting an asymptotic bound, these constants are unlikely to make a noticeable difference.
- **Simplifying Assumption 3:** Set $c = \max\{c_1, c_2\}$ and replace the equality with an upper bound.
- This is less exact, but is easier to manipulate.

The Final Recurrence

- Here is the final version of the recurrence we'll be working with:

$$\boxed{\begin{aligned}&\mathbf{T(1)} \le c\\&\mathbf{T(}n\mathbf{)} \le \mathbf{2T(}n\mathbf{ / 2) +} cn\end{aligned}}$$

- As before, we will justify why all of these simplifications are safe later on.
- The analysis we're about to do (without justifying the simplifications) is at the level we will expect for most of our discussion of divide-and-conquer algorithms.

Getting an Intuition

- Simple recurrence relations often give rise to surprising results.
- It is often useful to build up an intuition for what the recursion solves to before trying to formally prove it.
- We will explore two methods for doing this:
  - The *iteration method*.
  - The *recursion-tree method*.

Getting an Intuition

Simple recurrence relations often give rise to surprising results.

It is often useful to build up an intuition for what the recursion solves to before trying to formally prove it.

We will explore two methods for doing this:

- The *iteration method*.
- The *recursion-tree method*.

$$\boxed{\begin{aligned} &T(1) \le c \\ &T(n) \le 2T(n / 2) + cn \end{aligned}}$$

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

$T(1) \leq c$
$T(n) \leq 2T(n / 2) + cn$

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + c\,n \\ &\le 2\left(2T\left(\frac{n}{4}\right) + \frac{c\,n}{2}\right) + c\,n \end{aligned}$$

$$\boxed{\begin{aligned} &T(1) \le c \\ &T(n) \le 2T(n/2) + cn \end{aligned}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + c\,n \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{c\,n}{2}\right) + c\,n \\
&= 4T\left(\frac{n}{4}\right) + c\,n + c\,n
\end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n / 2) + cn \end{array}}$$

$$\boxed{\begin{array}{l}T(1) \le c \\ T(n) \le 2T(n / 2) + cn\end{array}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4T\left(\frac{n}{4}\right) + 2cn
\end{aligned}$$

$$\boxed{\begin{aligned}&T(1) \le c\\ &T(n) \le 2T(n/2) + cn\end{aligned}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn\\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn\\
&= 4T\left(\frac{n}{4}\right) + cn + cn\\
&= 4T\left(\frac{n}{4}\right) + 2cn\\
&\le 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn
\end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n/2) + cn \end{array}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4T\left(\frac{n}{4}\right) + 2cn \\
&\le 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 8T\left(\frac{n}{8}\right) + cn + 2cn
\end{aligned}$$

$$\boxed{\begin{array}{l}\mathrm{T}(1) \le c \\ \mathrm{T}(n) \le 2\mathrm{T}(n / 2) + cn\end{array}}$$

$$\begin{aligned}
\mathrm{T}(n) &\le 2\mathrm{T}\left(\frac{n}{2}\right)+cn \\
&\le 2\left(2\mathrm{T}\left(\frac{n}{4}\right)+\frac{cn}{2}\right)+cn \\
&= 4\mathrm{T}\left(\frac{n}{4}\right)+cn+cn \\
&= 4\mathrm{T}\left(\frac{n}{4}\right)+2cn \\
&\le 4\left(2\mathrm{T}\left(\frac{n}{8}\right)+\frac{cn}{4}\right)+2cn \\
&= 8\mathrm{T}\left(\frac{n}{8}\right)+cn+2cn \\
&= 8\mathrm{T}\left(\frac{n}{8}\right)+3cn
\end{aligned}$$

$$\boxed{\begin{aligned}&\mathrm{T}(1) \le c\\ &\mathrm{T}(n) \le 2\mathrm{T}(n/2) + cn\end{aligned}}$$

$$\begin{aligned}
\mathrm{T}(n) &\le 2\mathrm{T}\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2\mathrm{T}\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4\mathrm{T}\left(\frac{n}{4}\right) + cn + cn \\
&= 4\mathrm{T}\left(\frac{n}{4}\right) + 2cn \\
&\le 4\left(2\mathrm{T}\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 8\mathrm{T}\left(\frac{n}{8}\right) + cn + 2cn \\
&= 8\mathrm{T}\left(\frac{n}{8}\right) + 3cn \\
&\ldots \\
&\le 2^k\mathrm{T}\left(\frac{n}{2^k}\right) + kcn
\end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n/2) + cn \end{array}}$$

$$\boxed{n / 2^k = 1}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4T\left(\frac{n}{4}\right) + 2cn \\
&\le 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 8T\left(\frac{n}{8}\right) + cn + 2cn \\
&= 8T\left(\frac{n}{8}\right) + 3cn \\
&\dots \\
&\le 2^k T\left(\frac{n}{2^k}\right) + kcn
\end{aligned}$$

$$\boxed{\begin{array}{l}T(1) \le c \\ T(n) \le 2T(n / 2) + cn\end{array}}$$

$$\boxed{\begin{array}{c}n / 2^k = 1 \\ n = 2^k\end{array}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4T\left(\frac{n}{4}\right) + 2cn \\
&\le 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 8T\left(\frac{n}{8}\right) + cn + 2cn \\
&= 8T\left(\frac{n}{8}\right) + 3cn \\
&\ldots \\
&\le 2^k T\left(\frac{n}{2^k}\right) + kcn
\end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n/2) + cn \end{array}}$$

$$\boxed{\begin{array}{c} n / 2^k = 1 \\ n = 2^k \\ \log_2 n = k \end{array}}$$

$$\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4T\left(\frac{n}{4}\right) + 2cn \\
&\le 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 8T\left(\frac{n}{8}\right) + cn + 2cn \\
&= 8T\left(\frac{n}{8}\right) + 3cn \\
&\dots \\
&\le 2^k T\left(\frac{n}{2^k}\right) + kcn
\end{aligned}$$

$$\boxed{\begin{array}{l} \mathrm{T}(1) \le c \\ \mathrm{T}(n) \le 2\mathrm{T}(n / 2) + cn \end{array}}$$

$$\mathrm{T}(n) \quad \le \quad 2^k\,\mathrm{T}\left(\frac{n}{2^k}\right) + k\,c\,n$$

$$\boxed{\begin{aligned}&T(1) \le c\\&T(n) \le 2T(n/2) + cn\end{aligned}}$$

$$\begin{aligned}T(n) &\le 2^k\, T\!\left(\frac{n}{2^k}\right) + k\,c\,n\\&= 2^{\log_2 n}\, T(1) + c\,n\log_2 n\end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n / 2) + cn \end{array}}$$

$$\begin{aligned} T(n) &\le 2^k T\left(\frac{n}{2^k}\right) + k\,c\,n \\ &= 2^{\log_2 n} T(1) + c\,n \log_2 n \\ &= n\,T(1) + c\,n \log_2 n \end{aligned}$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n / 2) + cn \end{array}}$$

$$\begin{aligned} T(n) &\le 2^k T\left(\frac{n}{2^k}\right) + k\,c\,n \\ &= 2^{\log_2 n} T(1) + c\,n\log_2 n \\ &= n\,T(1) + c\,n\log_2 n \\ &\le c\,n + c\,n\log_2 n \end{aligned}$$

$$\boxed{\begin{aligned}&T(1) \le c\\&T(n) \le 2T(n / 2) + cn\end{aligned}}$$

$$\begin{aligned}
T(n) &\le 2^k T\left(\frac{n}{2^k}\right) + kcn\\
&= 2^{\log_2 n} T(1) + cn\log_2 n\\
&= nT(1) + cn\log_2 n\\
&\le cn + cn\log_2 n\\
&= O(n\log n)
\end{aligned}$$

The Iteration Method

- What we just saw is an example of the *iteration method*.
- Keep plugging the recurrence into itself until you spot a pattern, then try to simplify.
- Doesn't always give an exact answer, but useful for building up an intuition.

Getting an Intuition

- Simple recurrence relations often give rise to surprising results.
- It is often useful to build up an intuition for what the recursion solves to before trying to formally prove it.
- We will explore two methods for doing this:
  - The *iteration method*.
  - The *recursion-tree method*.

Getting an Intuition

Simple recurrence relations often give rise to surprising results.

It is often useful to build up an intuition for what the recursion solves to before trying to formally prove it.

We will explore two methods for doing this:

- The *iteration method*.
- The *recursion-tree method*.

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n / 2) + cn \end{array}}$$

$$\boxed{\begin{array}{l} T(1) \leq c \\ T(n) \leq 2T(n / 2) + cn \end{array}}$$

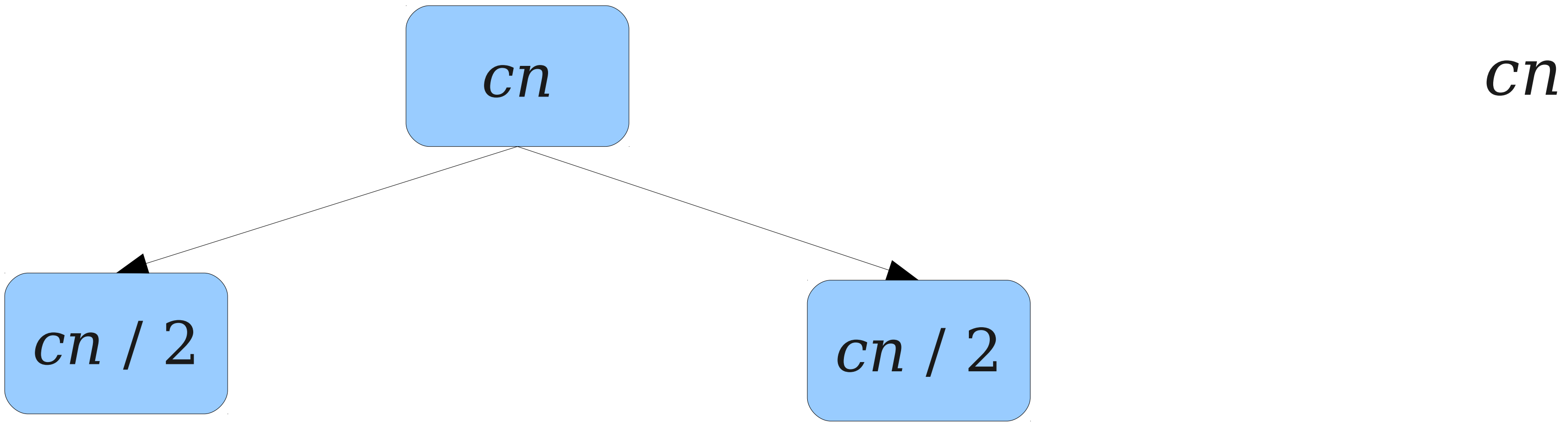
cn

$$T(1) \leq c$$
$$T(n) \leq 2T(n / 2) + cn$$

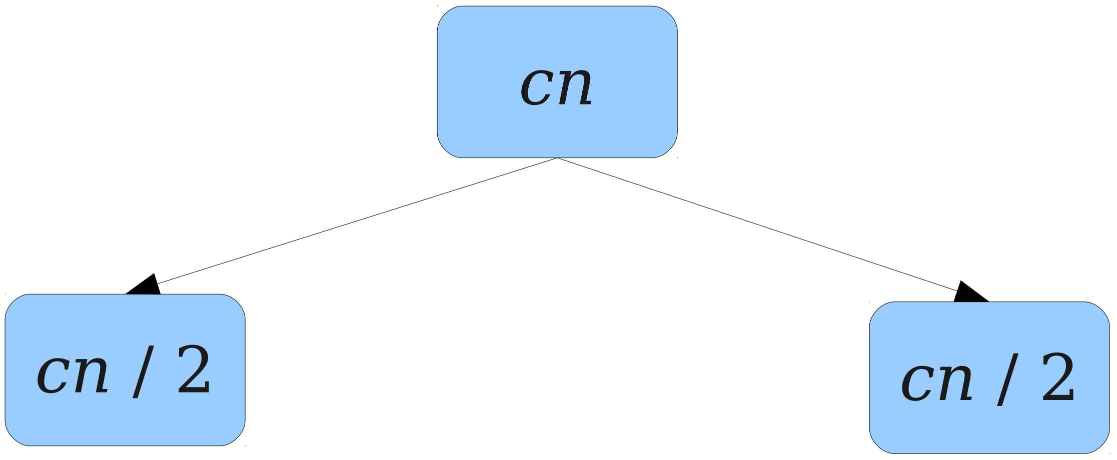
cn

cn

$$T(1) \leq c$$
$$T(n) \leq 2T(n / 2) + cn$$



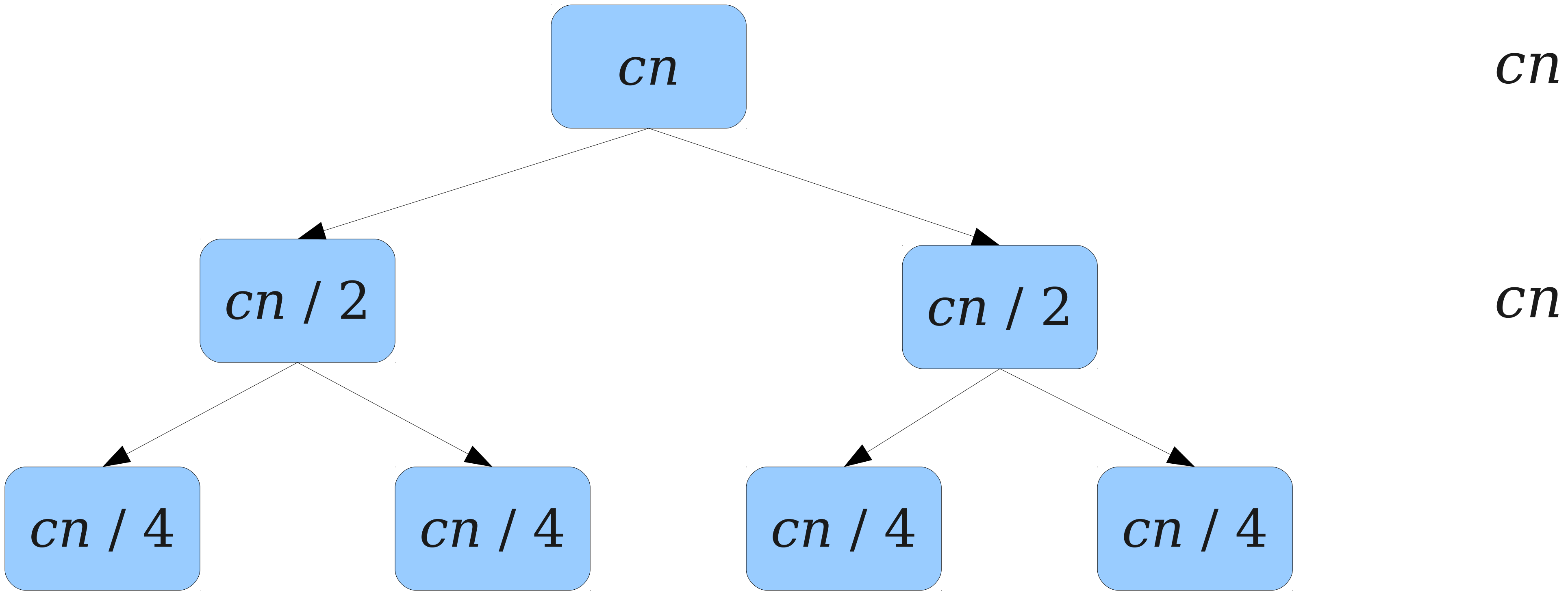
$$T(1) \le c$$
$$T(n) \le 2T(n / 2) + cn$$



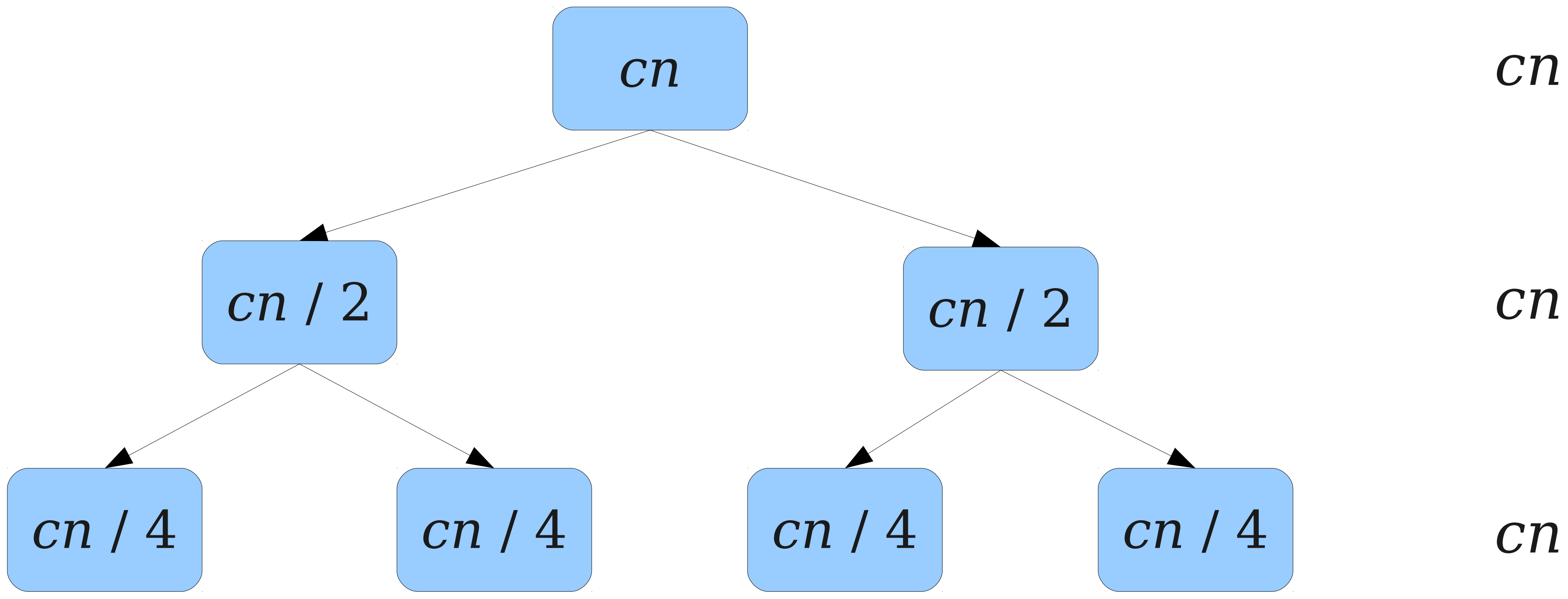
cn

cn

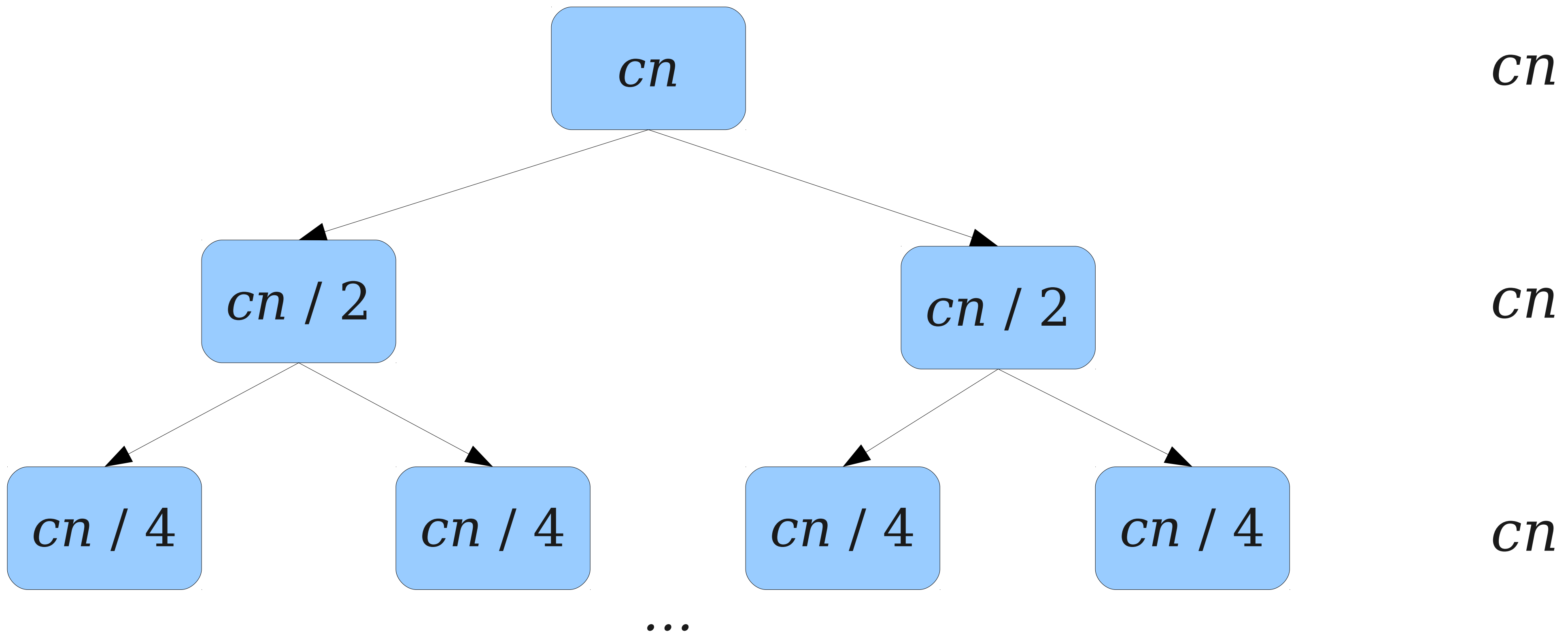
$$T(1) \le c$$
$$T(n) \le 2T(n / 2) + cn$$



$$T(1) \le c$$
$$T(n) \le 2T(n / 2) + cn$$

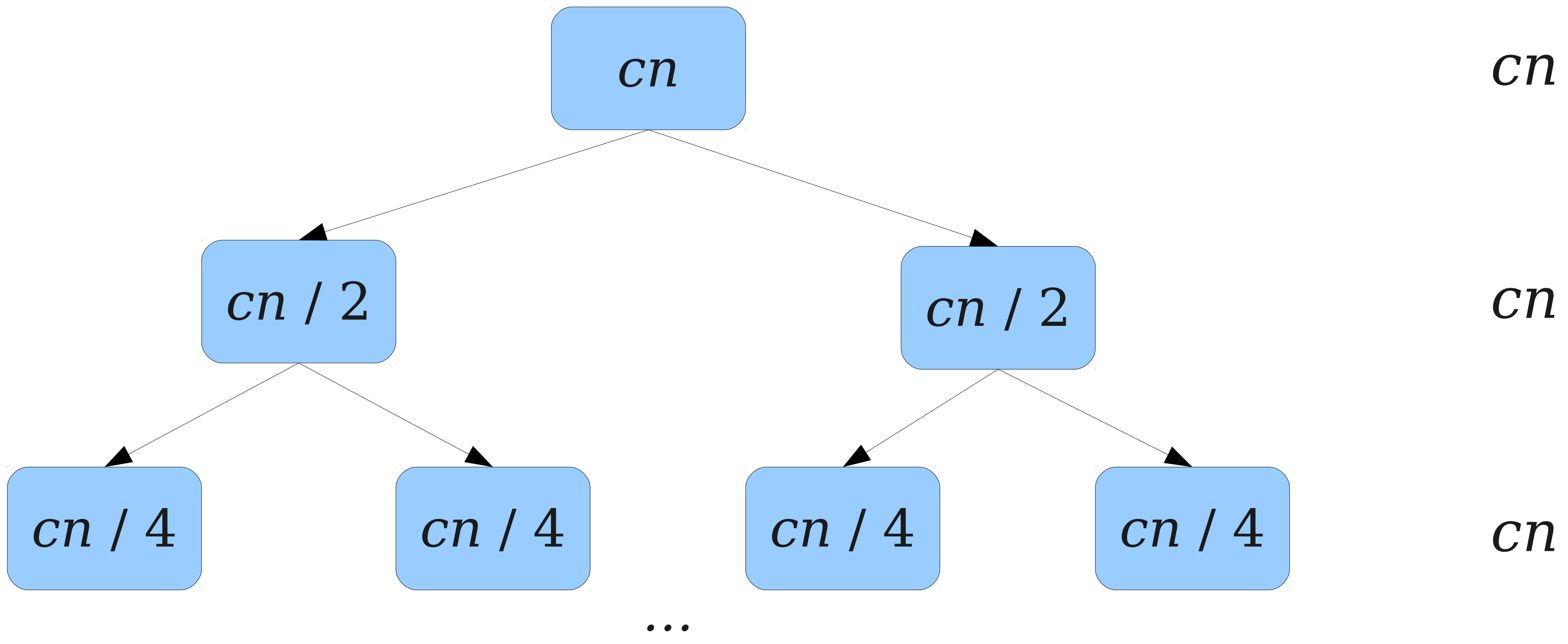


$T(1) \le c$
$T(n) \le 2T(n / 2) + cn$



$T(1) \le c$

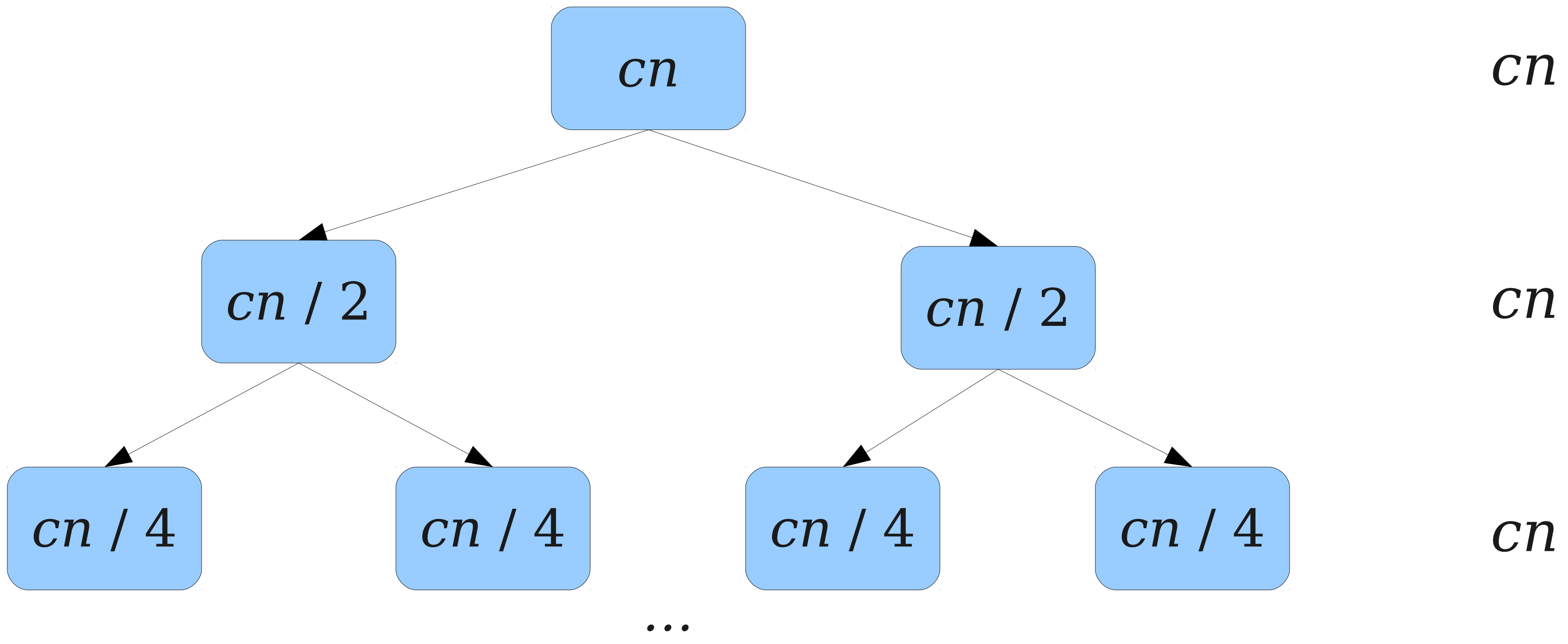
$T(n) \le 2T(n / 2) + cn$



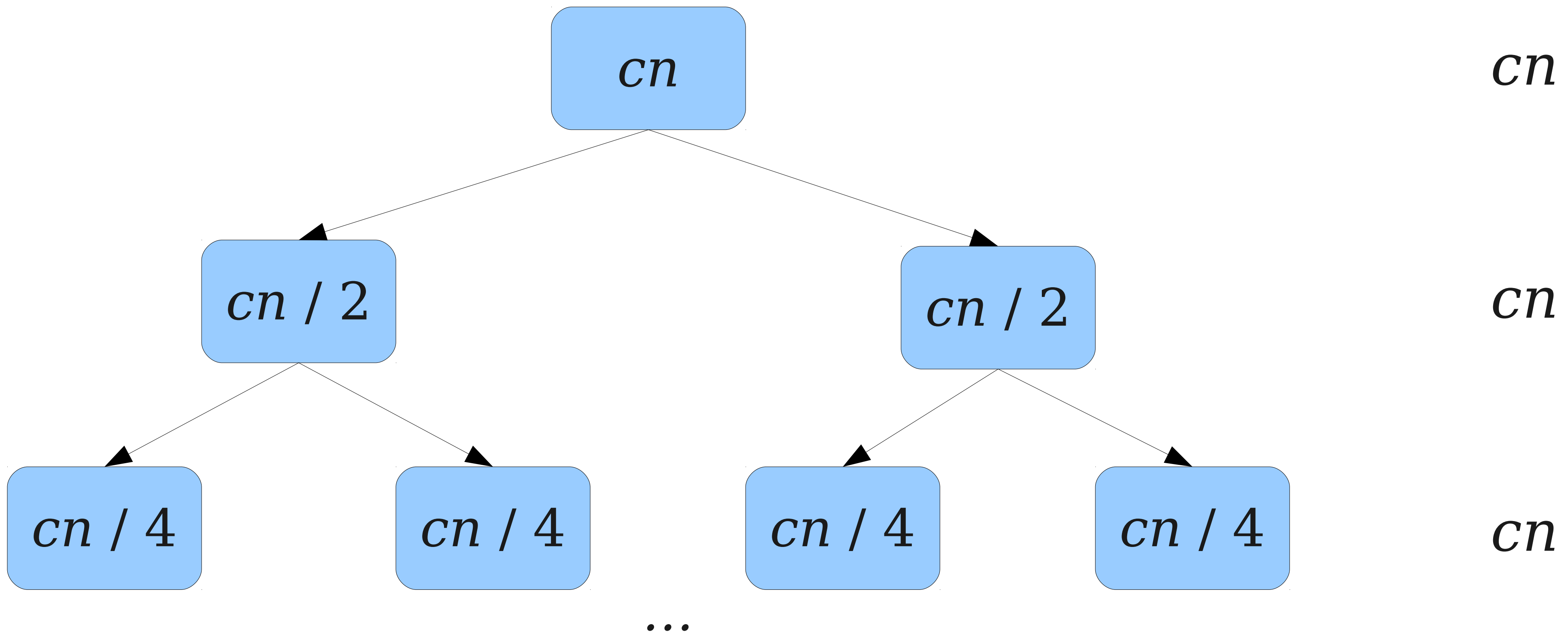
There are **$\log_2 n + 1$** layers in the tree (numbered 0, 1, 2, …, $\log_2 n$).

The first $\log_2 n$ of them are the recursive case. The last one consists purely of base cases.

$$T(1) \le c$$
$$T(n) \le 2T(n / 2) + cn$$

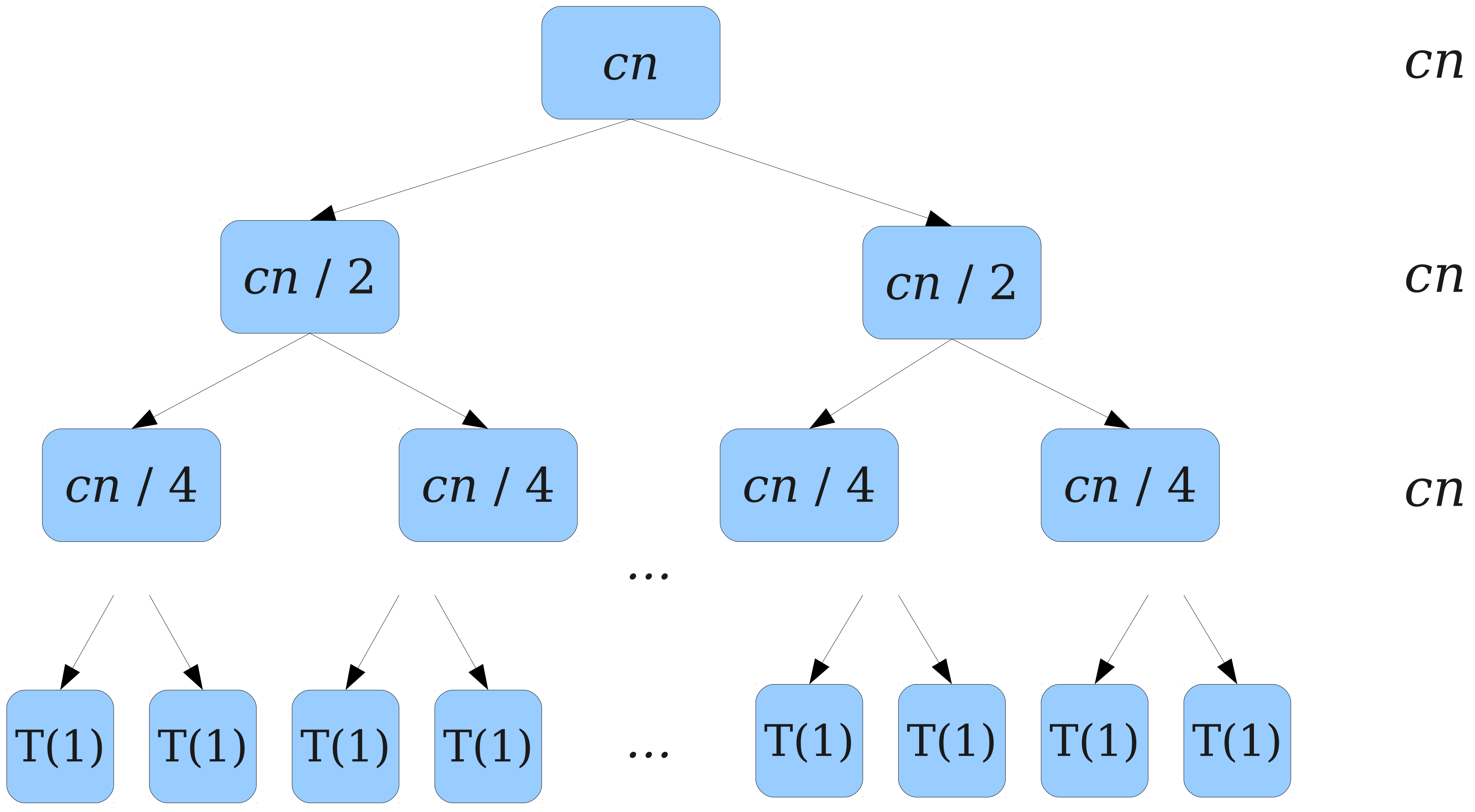


$T(1) \le c$
$T(n) \le 2T(n / 2) + cn$



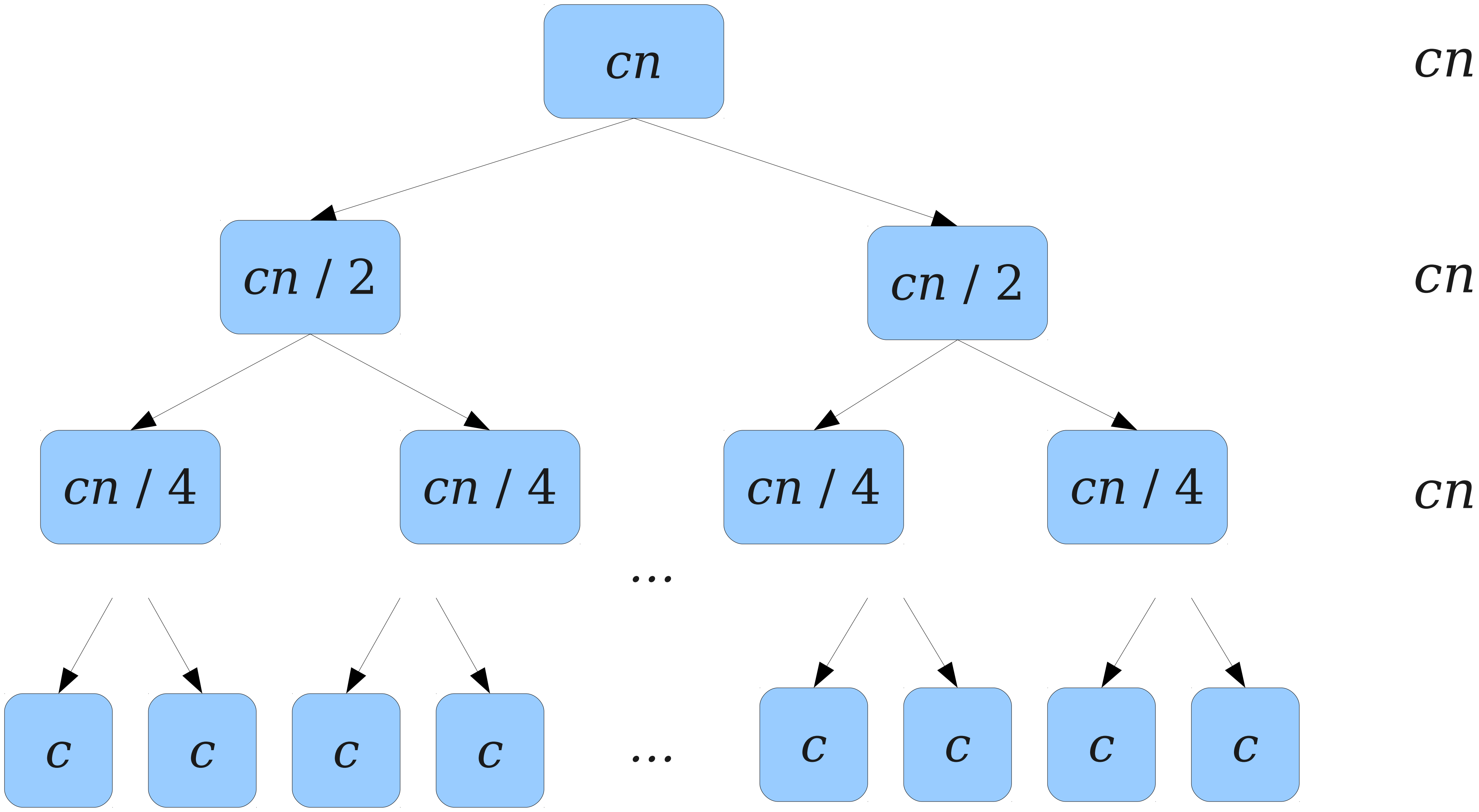
$cn \log_2 n$

$T(1) \le c$
$T(n) \le 2T(n / 2) + cn$



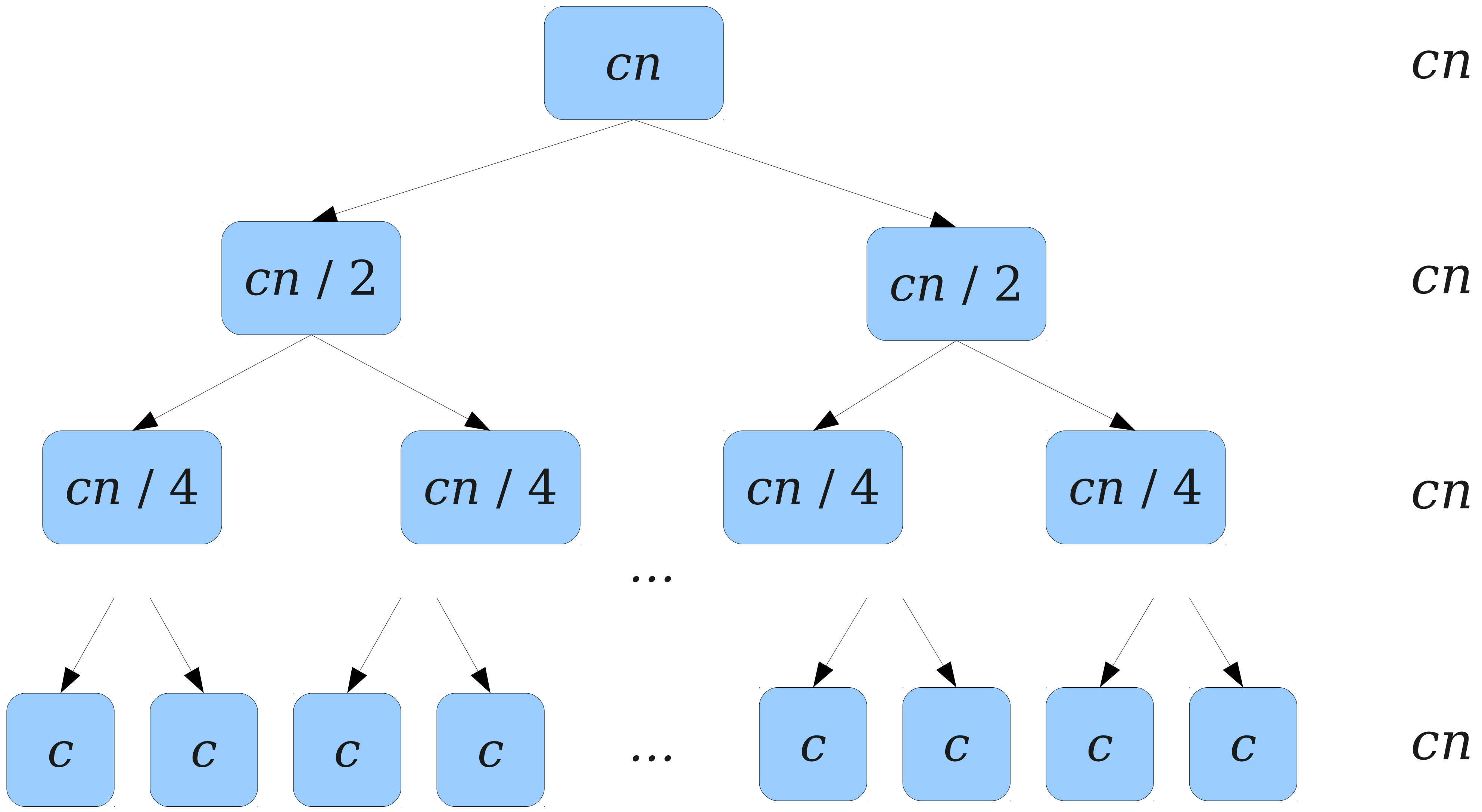
$cn \log_2 n$

$$T(1) \le c$$
$$T(n) \le 2T(n / 2) + cn$$



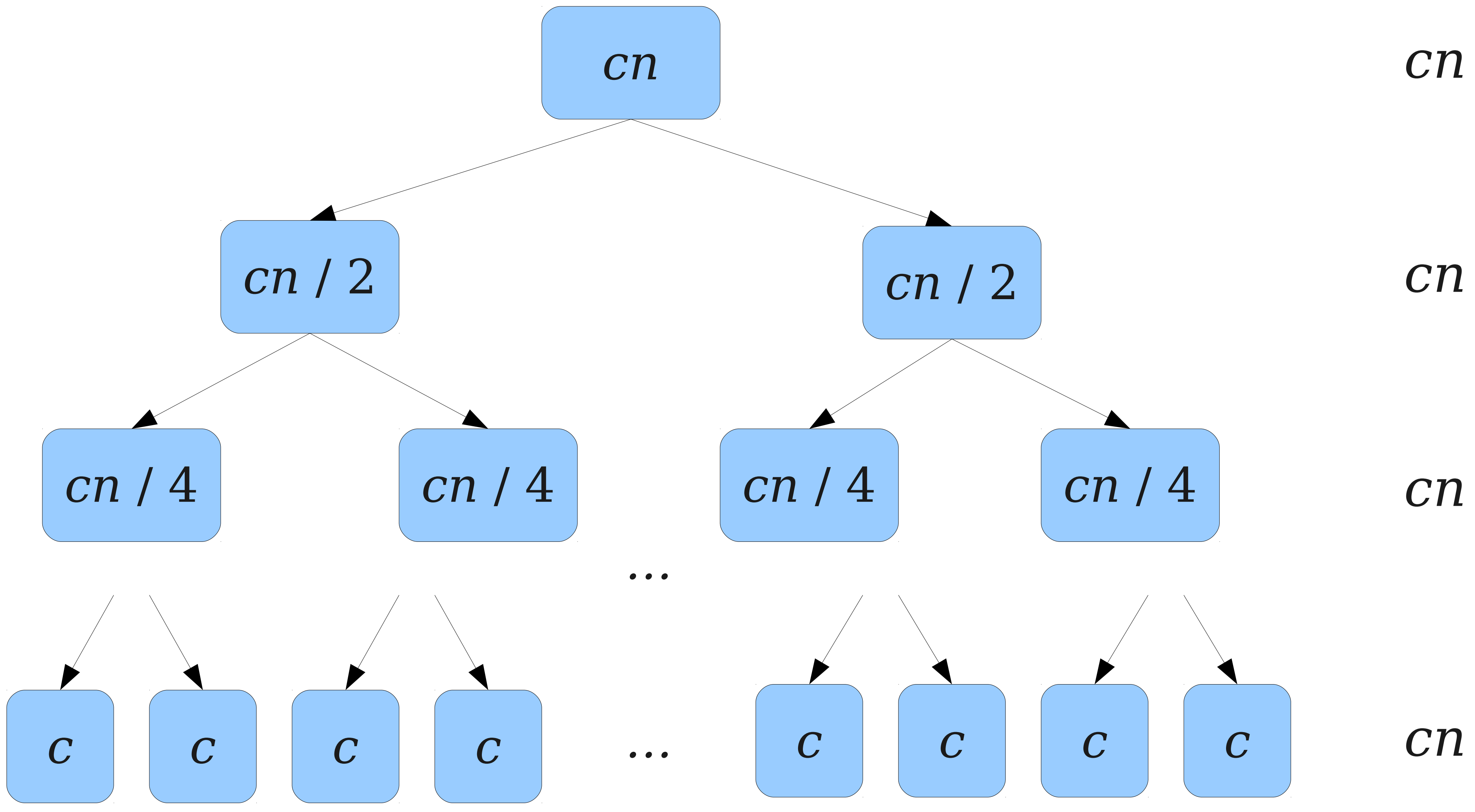
$$cn \log_2 n$$

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le 2T(n/2) + cn \end{array}}$$



$cn \log_2 n$

$T(1) \le c$
$T(n) \le 2T(n / 2) + cn$



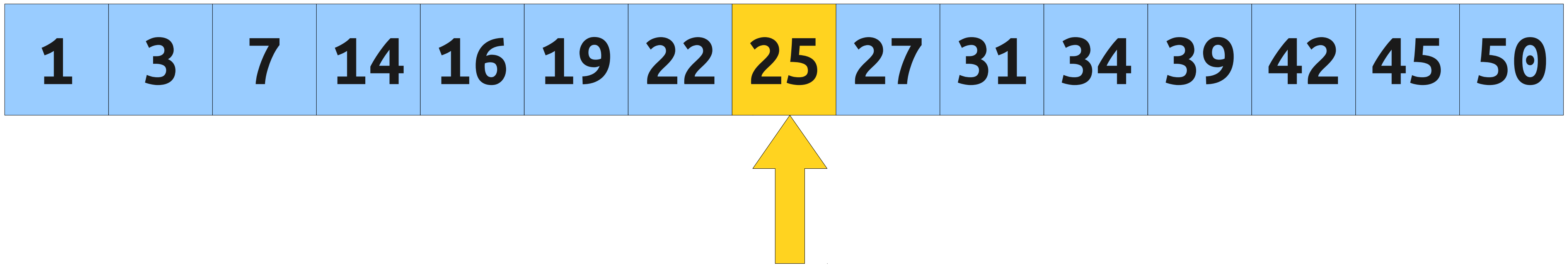
$cn \log_2 n + cn$

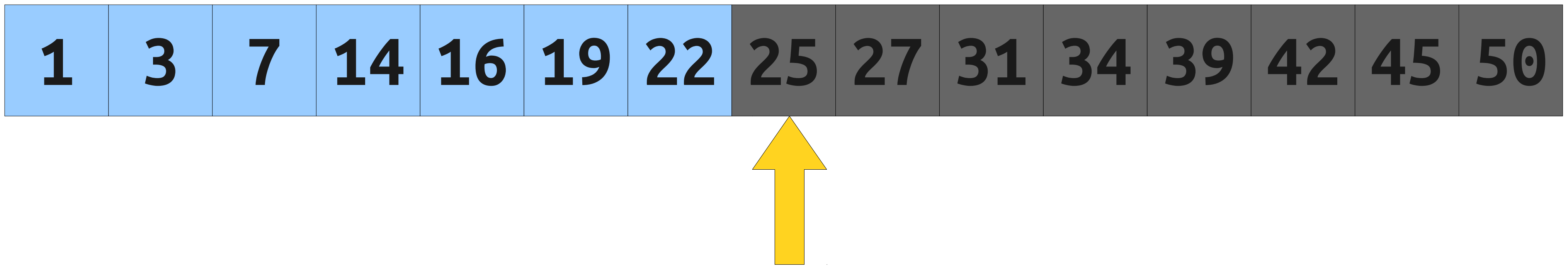
The Recursion Tree Method

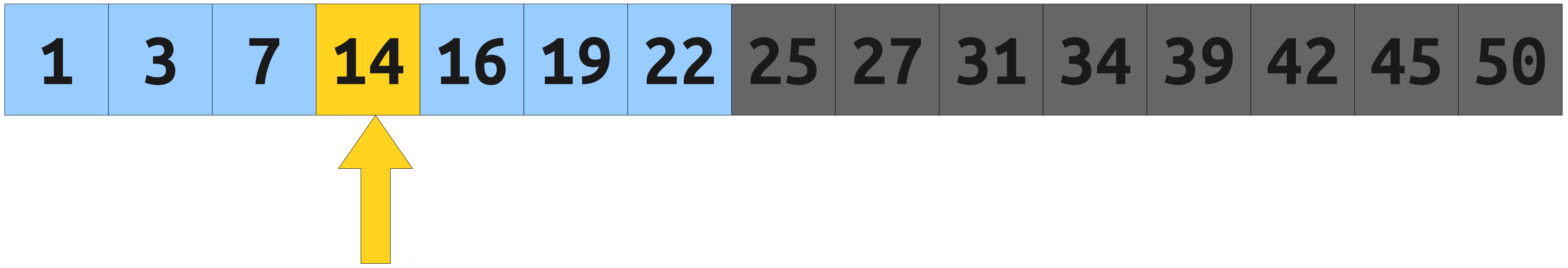
- This diagram is called a **recursion tree** and accounts for how much total work each recursive call makes.
- Often useful to sum up the work across the layers of the tree.

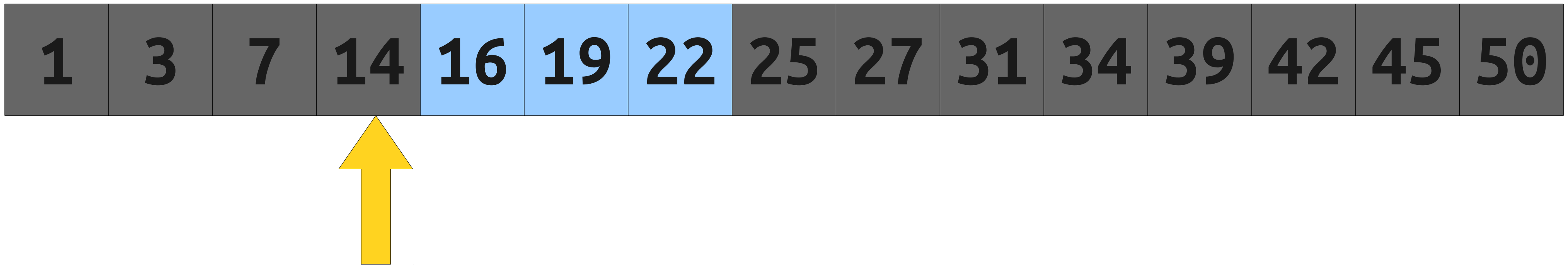
Another Algorithm: **Binary Search**

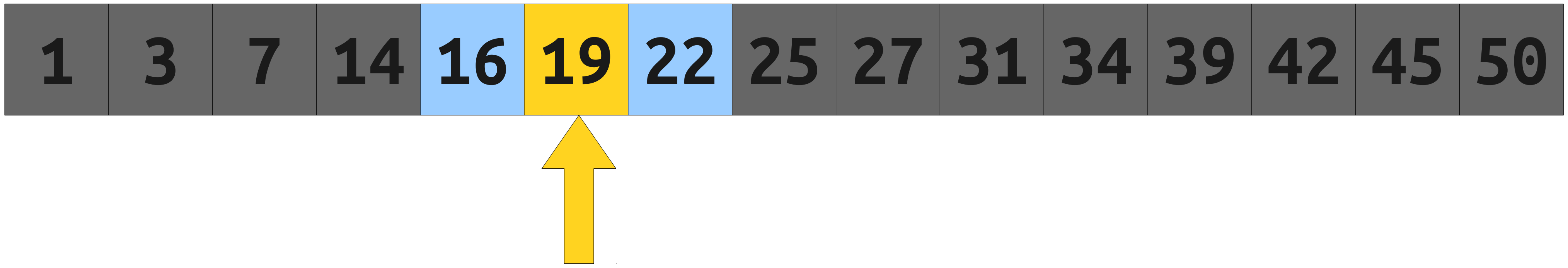
1	3	7	14	16	19	22	25	27	31	34	39	42	45	50

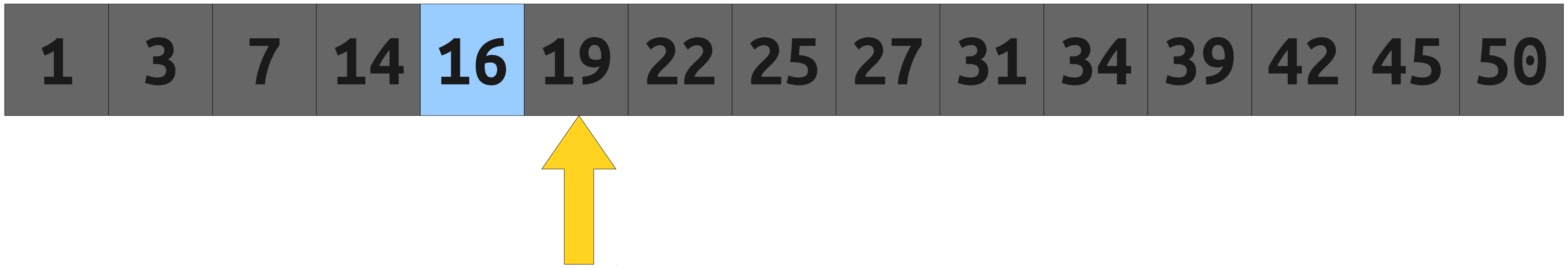

1
3
7
14
16
19
22
25
27
31
34
39
42
45
50

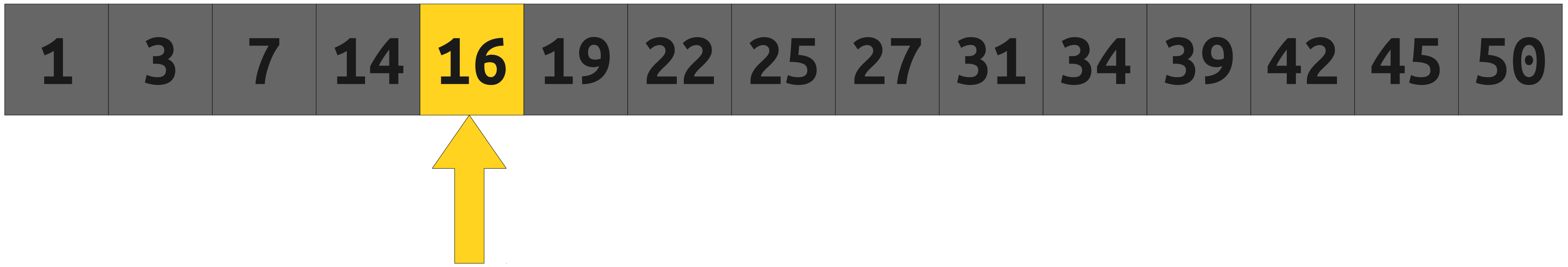

1
3
7
14
16
19
22
25
27
31
34
39
42
45
50

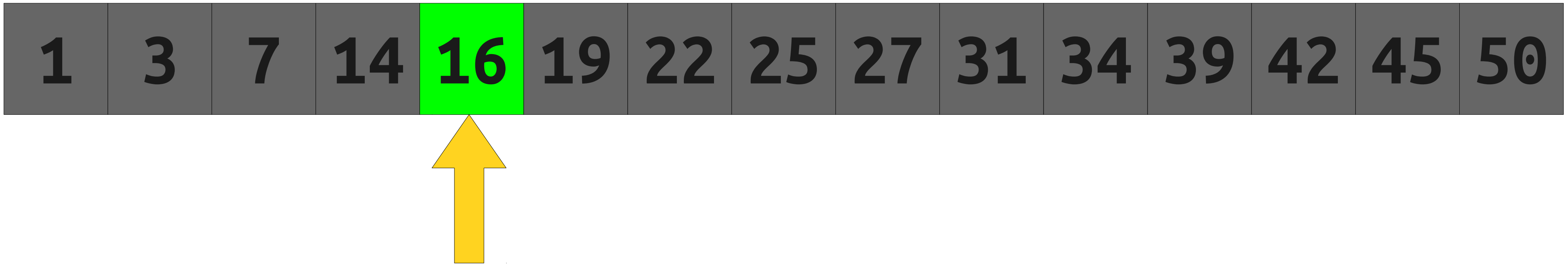

1
3
7
14
16
19
22
25
27
31
34
39
42
45
50


1
3
7
14
16
19
22
25
27
31
34
39
42
45
50


1
3
7
14
16
19
22
25
27
31
34
39
42
45
50


1
3
7
14
16
19
22
25
27
31
34
39
42
45
50


1
3
7
14
16
19
22
25
27
31
34
39
42
45
50


1
3
7
14
16
19
22
25
27
31
34
39
42
45
50

```
procedure binarySearch(list A, int low, int high,
                                 value key):
  if low ≥ high:
     return false

  let mid = ⌊(high + low) / 2⌋
  if A[mid] = key:
     return true
  else if A[mid] > key:
     return binarySearch(a, low, mid)
  else (A[mid] < key):
     return binarySearch(a, mid + 1, high)
```

```
procedure binarySearch(list A, int low, int high,
                       value key):
  if low ≥ high:
     return false

  let mid = ⌊(high + low) / 2⌋
  if A[mid] = key:
     return true
  else if A[mid] > key:
     return binarySearch(a, low, mid)
  else (A[mid] < key):
     return binarySearch(a, mid + 1, high)
```

$T(0) = \Theta(1)$
$T(1) = \Theta(1)$

```
procedure binarySearch(list A, int low, int high,
                       value key):
  if low ≥ high:
    return false

  let mid = ⌊(high + low) / 2⌋
  if A[mid] = key:
    return true
  else if A[mid] > key:
    return binarySearch(a, low, mid)
  else (A[mid] < key):
    return binarySearch(a, mid + 1, high)
```

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) \leq T(\lfloor n / 2 \rfloor) + \Theta(1)$$

```
procedure binarySearch(list A, int low, int high,
                       value key):
  if low ≥ high:
     return false

  let mid = ⌊(high + low) / 2⌋
  if A[mid] = key:
     return true
  else if A[mid] > key:
     return binarySearch(a, low, mid)
  else (A[mid] < key):
     return binarySearch(a, mid + 1, high)
```

$$T(1) \leq c$$
$$T(n) \leq T(n / 2) + c$$

The Iteration Method

$$\boxed{\begin{array}{l} T(1) \leq c \\ T(n) \leq T(n / 2) + c \end{array}}$$

The Iteration Method

T(1) ≤ c
T(n) ≤ T(n / 2) + c

$$\mathrm{T}(n) \le T\left(\frac{n}{2}\right) + c$$

The Iteration Method

T(1) ≤ c
T(n) ≤ T(n / 2) + c

$$
\begin{aligned}
\mathrm{T}(n) &\le T\left(\frac{n}{2}\right)+c \\
&\le \left(T\left(\frac{n}{4}\right)+c\right)+c
\end{aligned}
$$

The Iteration Method

$T(1) \le c$
$T(n) \le T(n / 2) + c$

$$\begin{aligned} \mathrm{T}(n) &\le T\left(\frac{n}{2}\right)+c \\ &\le \left(T\left(\frac{n}{4}\right)+c\right)+c \\ &= T\left(\frac{n}{4}\right)+2c \end{aligned}$$

The Iteration Method

T(1) ≤ c
T(n) ≤ T(n / 2) + c

$$\begin{aligned} \mathrm{T}(n) &\le T\left(\frac{n}{2}\right)+c \\ &\le \left(T\left(\frac{n}{4}\right)+c\right)+c \\ &= T\left(\frac{n}{4}\right)+2c \\ &\le \left(T\left(\frac{n}{8}\right)+c\right)+2c \end{aligned}$$

The Iteration Method

T(1) ≤ c
T(n) ≤ T(n / 2) + c

$$\begin{aligned}
\mathrm{T}(n) &\le T\left(\frac{n}{2}\right)+c \\
&\le \left(T\left(\frac{n}{4}\right)+c\right)+c \\
&= T\left(\frac{n}{4}\right)+2c \\
&\le \left(T\left(\frac{n}{8}\right)+c\right)+2c \\
&= T\left(\frac{n}{8}\right)+3c
\end{aligned}$$

The Iteration Method

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le T(n/2) + c \end{array}}$$

$$\begin{aligned}
T(n) &\le T\left(\frac{n}{2}\right) + c \\
&\le \left(T\left(\frac{n}{4}\right) + c\right) + c \\
&= T\left(\frac{n}{4}\right) + 2c \\
&\le \left(T\left(\frac{n}{8}\right) + c\right) + 2c \\
&= T\left(\frac{n}{8}\right) + 3c \\
&\dots \\
&\le T\left(\frac{n}{2^k}\right) + kc
\end{aligned}$$

The Iteration Method

$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le T(n / 2) + c \end{array}}$$

$$T(n) \le T\left(\frac{n}{2^k}\right) + k\,c$$

The Iteration Method

$\mathrm{T}(1) \le c$
$\mathrm{T}(n) \le \mathrm{T}(n / 2) + c$

$$\begin{aligned} \mathrm{T}(n) &\le T\left(\frac{n}{2^k}\right) + k\,c \\ &= T(1) + c\log_2 n \end{aligned}$$

The Iteration Method

T(1) ≤ c
T(n) ≤ T(n / 2) + c

$$
\begin{aligned}
\mathrm{T}(n) &\le T\left(\frac{n}{2^k}\right) + k\,c \\
&= T(1) + c\log_2 n \\
&\le c + c\log_2 n
\end{aligned}
$$

The Iteration Method

$$\boxed{\begin{array}{l}T(1) \le c \\ T(n) \le T(n/2) + c\end{array}}$$

$$\begin{aligned} T(n) &\le T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + c\log_2 n \\ &\le c + c\log_2 n \\ &= O(\log n) \end{aligned}$$

The Recursion Tree Method

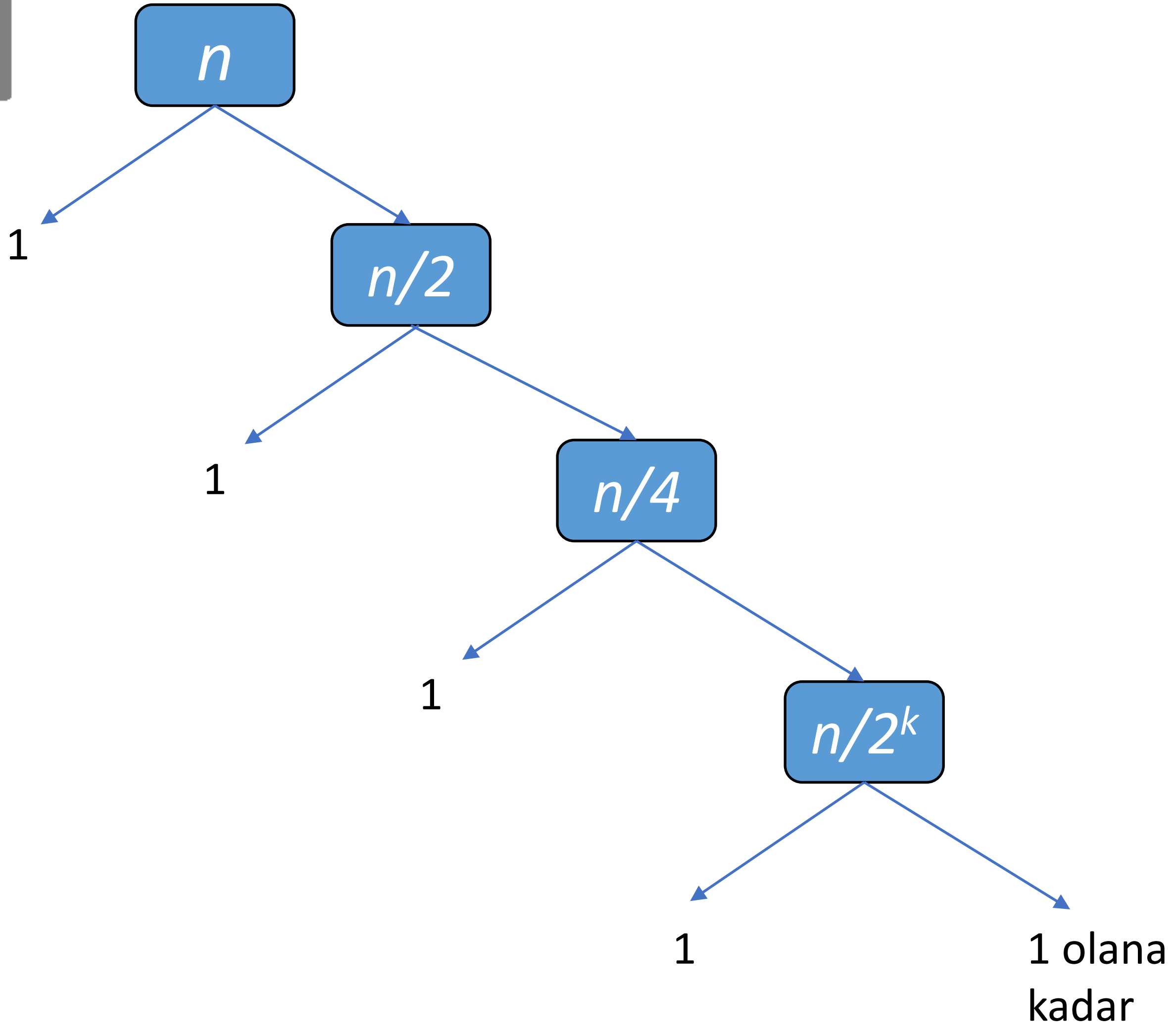
$$\boxed{\begin{array}{l} T(1) \le c \\ T(n) \le T(n / 2) + c \end{array}}$$

$$T(n) = ck$$

$$T\left(\frac{n}{2^k}\right) = 1$$

$$k = log_2 n$$

$$T(n) = clog_2 n$$


n
1
n/2
1
n/4
1
n/2^k
1
1 olana kadar