طراحی الگوریتم ها

مبحث پنجم:
تقسیم و غلبه و مرتب سازی ادغامی

سجاد شیرعلی شهرضا

بهار 1402

یکشنبه، 30 بهمن 1401

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

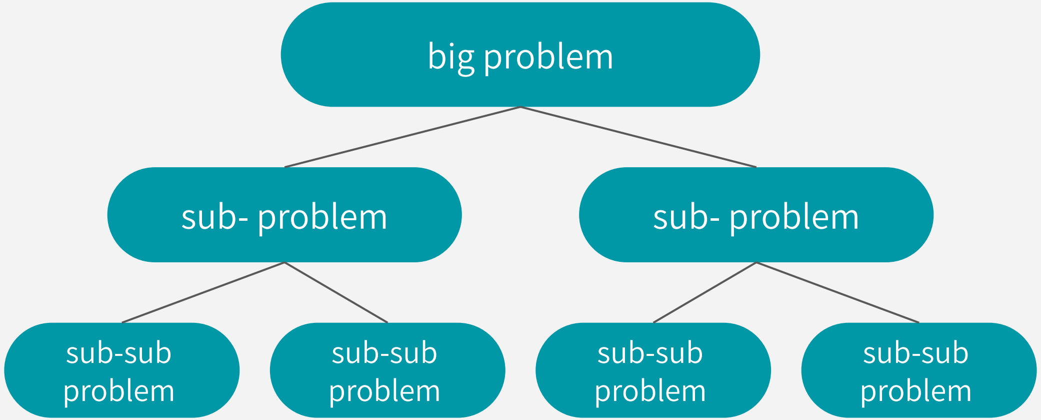
- بخش مرتبط کتاب برای این جلسه: 2.1، 2.3
- امتحانک اول
  - یکشنبه هفته آینده، 7 اسفند
  - به صورت حضوری در کلاس
  - در ساعت کلاس
  - در صورت تغییر، از طریق سایت اطلاع رسانی خواهد شد.

تقسیم و غلبه

یک روش پایه در طراحی الگوریتم

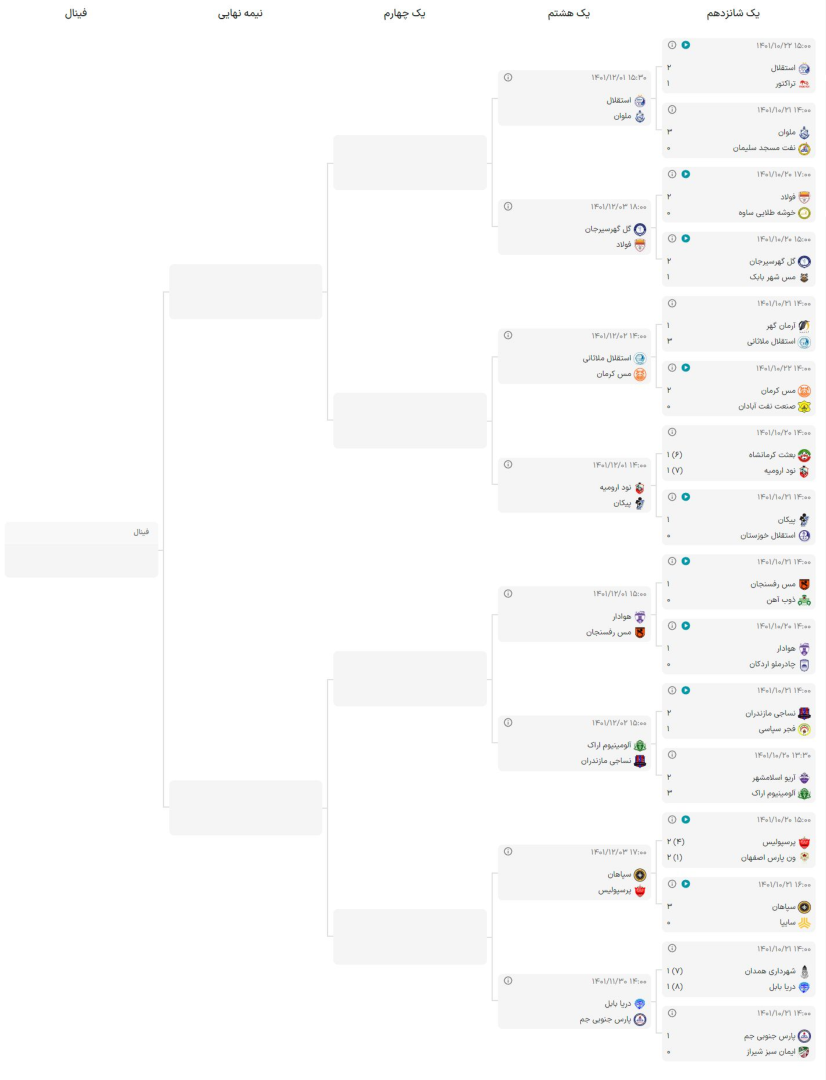
DIVIDE AND CONQUER

- **An algorithm design paradigm:**
  1. break up a problem into smaller subproblems
  2. solve those subproblems *recursively*
  3. combine the results of those subproblems to get the overall answer



مثال 1: بازی های جام حذفی

- هدف: یافتن قهرمان
- تقسیم تیم ها به دو گروه
- یافتن قهرمان هر گروه
- مسابقه بین دو قهرمان گروهی
    - تعیین قهرمان کلی



EXAMPLE 2: MULTIPLICATION

- **Original large problem:** multiply two 4-digit numbers
- **What are the subproblems?** Let's unravel some stuff...

$$1234 \times 5678$$

$$= (12\text{x}100 + 34) \text{ x } (56\text{x}100 + 78)$$

$$= (12\text{x}56)100^2 + (12\text{x}78 + 34\text{x}56)100 + (34\text{x}78)$$

MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply two 4-digit numbers
- **What are the subproblems?** Let's unravel some stuff...

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (\underbrace{12 \times 56}_{1})100^2 + (\underbrace{12 \times 78}_{2} + \underbrace{34 \times 56}_{3})100 + (\underbrace{34 \times 78}_{4})$$

One 4-digit problem → *Four 2-digit subproblems*

MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply 2 n-digit numbers
- **What are the subproblems?** More generally:

$$[x_1 \ldots x_{n/2} x_{n/2+1} \ldots x_n] \times [y_1 \ldots y_{n/2} y_{n/2+1} \ldots y_n]$$

$$= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (a \times c)10^n + (a \times d + b \times c)10^{n/2} + (b \times d)$$

1 2 3 4

One n-digit problem → *Four (n/2)-digit subproblems*

EXAMPLE 3: THE SORTING TASK

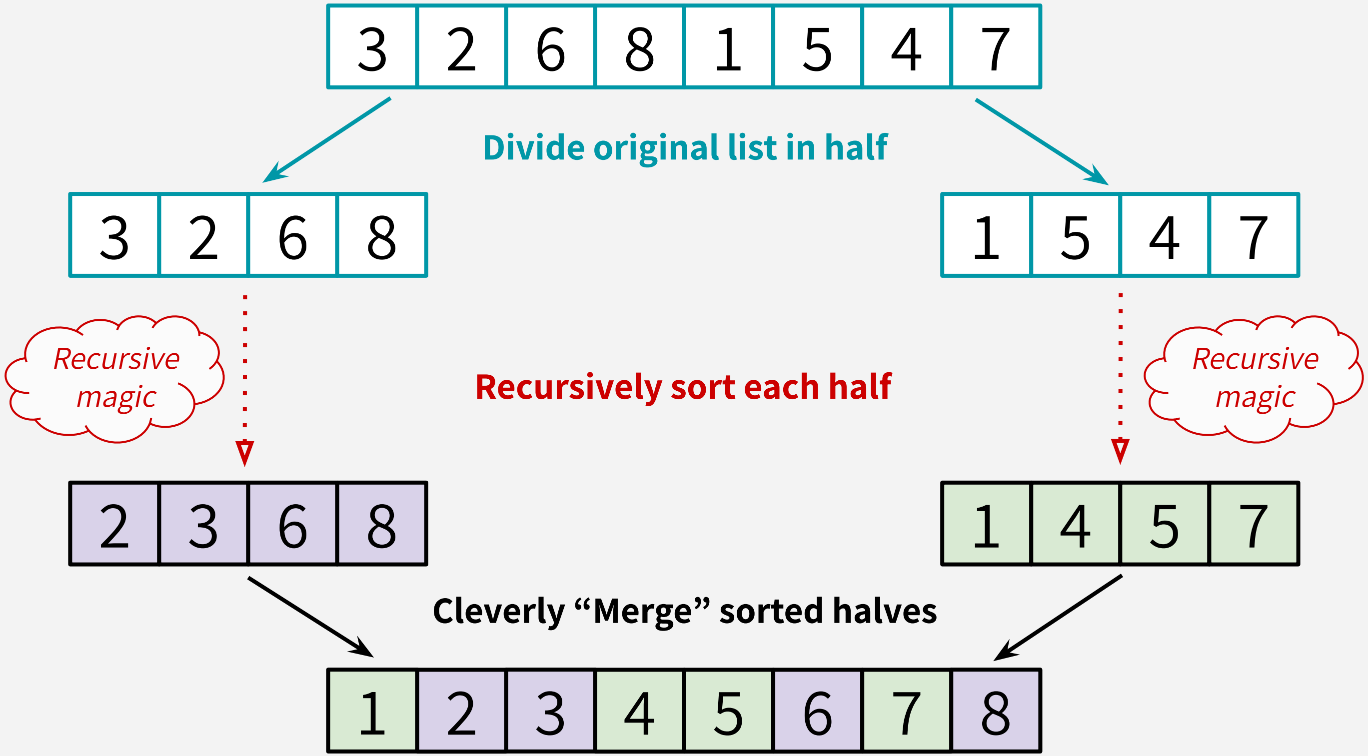
INPUT: a list of n elements (for today, we'll assume all elements are distinct)

3	2	6	8	1	5	4	7

1	2	3	4	5	6	7	8

OUTPUT: a list with those n elements in sorted order!

MERGESORT IDEA



MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
   n = len(A)
   if n <= 1:
      return A
   L = MERGESORT(A[0:n/2])
   R = MERGESORT(A[n/2:n])
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

For today, let's assume that n is a power of 2.

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

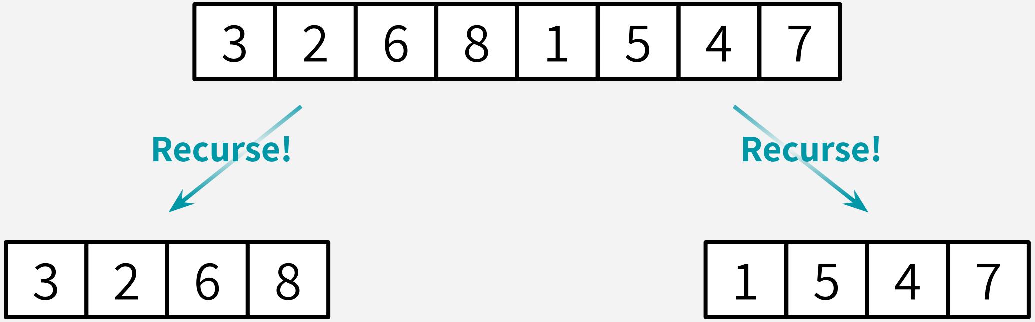
```
MERGE*(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

* Not complete! Some corner cases are missing.

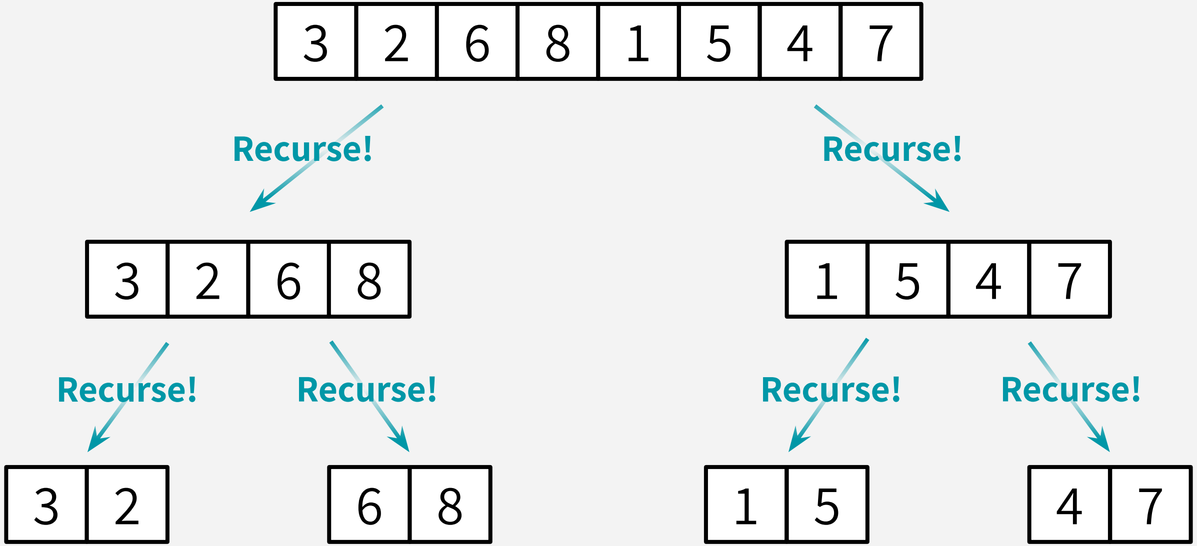
MERGESORT: RECURSIVE CALLS

3	2	6	8	1	5	4	7

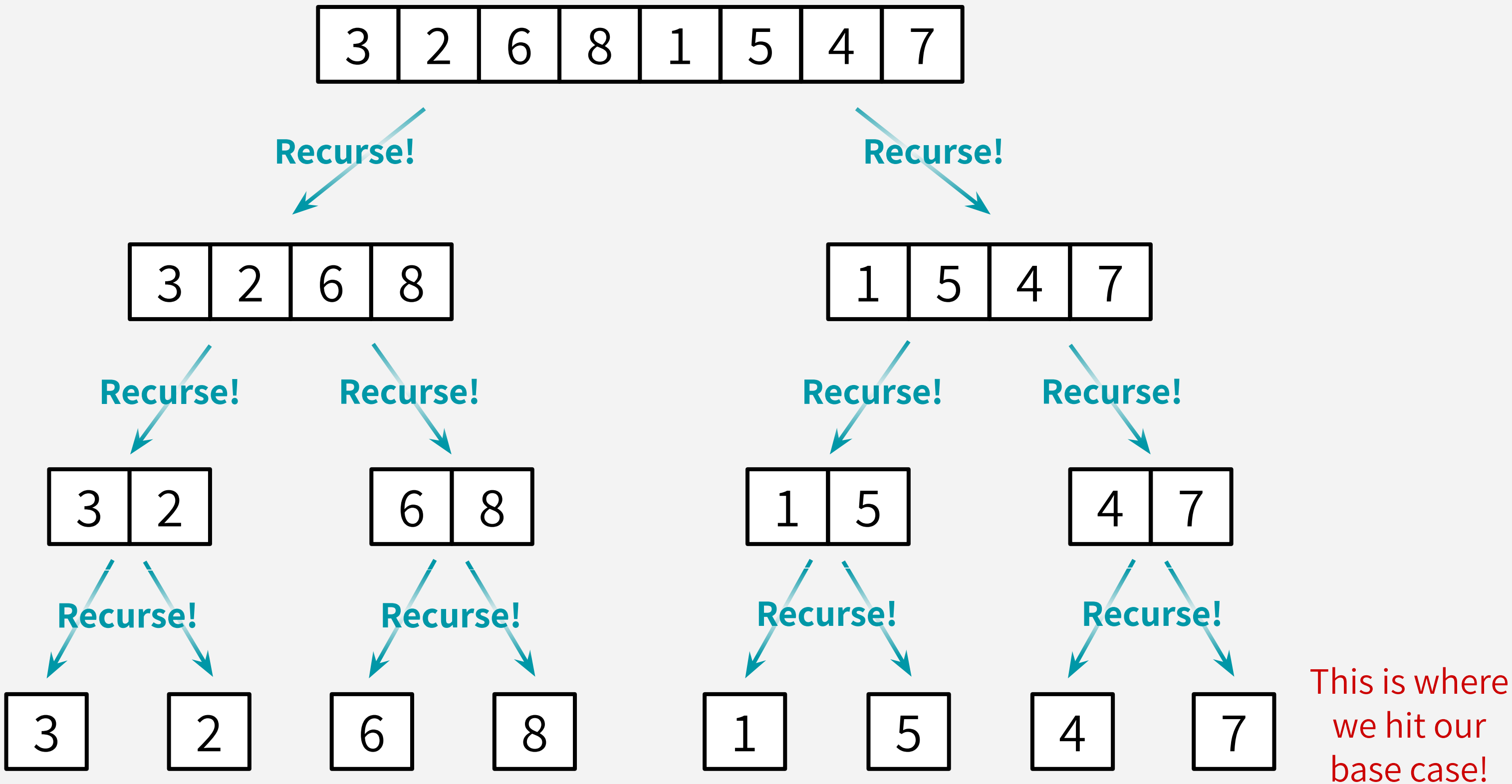
MERGESORT: RECURSIVE CALLS



MERGESORT: RECURSIVE CALLS



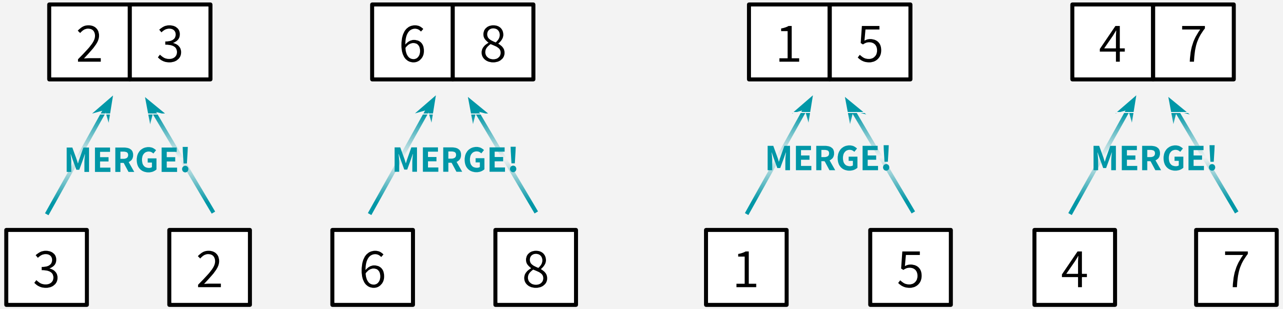
MERGESORT: RECURSIVE CALLS



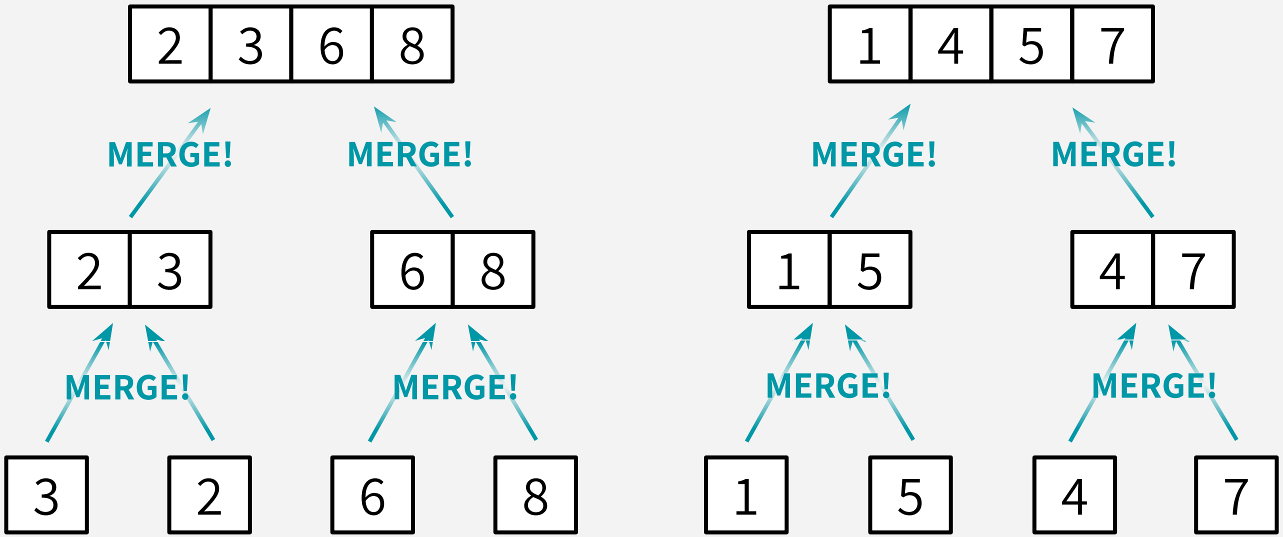
MERGESORT: MERGE STEPS

3	2	6	8	1	5	4	7

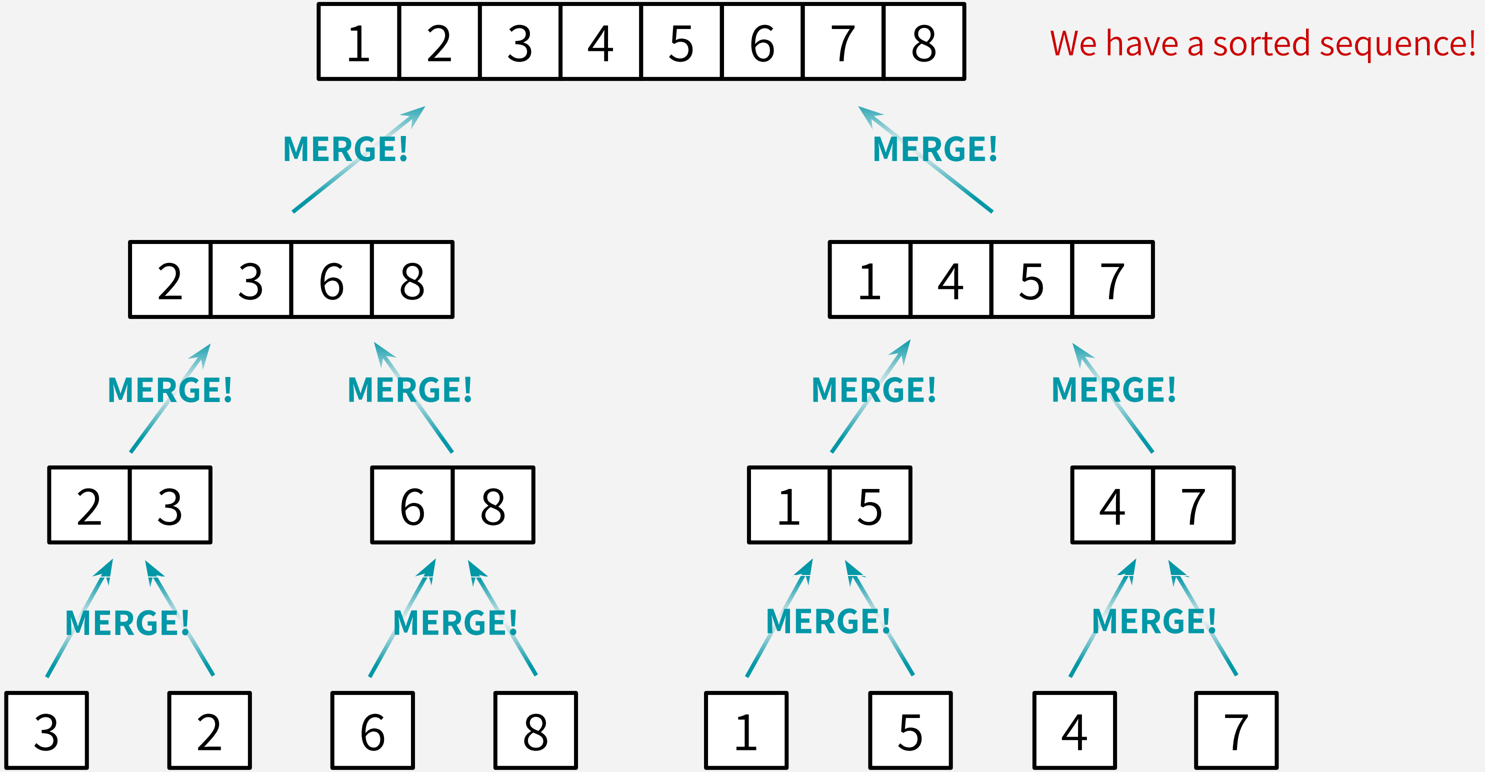
MERGESORT: MERGE STEPS



MERGESORT: MERGE STEPS



MERGESORT: MERGE STEPS



سوال؟

اثبات درستی مرتب سازی ادغامی

آیا واقعا ورودی را مرتب میکند؟

MERGESORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Whenever we make two "child" recursive calls, as long as those calls successfully sort our left and right halves, we'll safely merge them to create a fully sorted array.

In other words: as long as our recursive calls work on arrays of <u>*smaller*</u> *lengths, then our algorithm will correctly return a sorted array.*

MERGESORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Whenever we make two "child" recursive calls, as long as those calls successfully sort our left and right halves, we'll safely merge them to create a fully sorted array.

In other words: as long as our recursive calls work on arrays of <u>smaller</u> lengths, then our algorithm will correctly return a sorted array.

THIS IS A JOB FOR: ***PROOF BY INDUCTION!***

(Here, we perform induction on the *length of input list*)

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

INDUCTIVE STEP *(weak induction version)*

Next, assume that the inductive hypothesis holds when **i** takes on some value **k**.
Now prove that the IH holds as well when **i** takes on the value **k+1**.

CONCLUSION

By induction, conclude that the IH holds across the range of **i** you're dealing with.

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations ⇒ yay!

RECURSIVE ALGORITHMS

1. **Inductive hypothesis**: your algorithm is correct for sizes *up to* **i**
2. **Base case**: IH holds for i < small constant
3. **Inductive step**:
   - assume IH holds for k ⇒ prove k+1, *OR*
   - assume IH holds for {1,2,...,k-1} ⇒ prove k.
4. **Conclusion**: IH holds for i = n ⇒ yay!

MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

In every recursive call on an array of length *at most* **i**, MERGESORT returns a sorted array.

MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

In every recursive call on an array of length *at most* **i**, MERGESORT returns a sorted array.

BASE CASE

The IH holds for i = 1: A 1-element array is always sorted.

MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

In every recursive call on an array of length *at most* **i**, MERGESORT returns a sorted array.

BASE CASE

The IH holds for i = 1: A 1-element array is always sorted.

INDUCTIVE STEP *(strong/complete induction)*

Let k be an integer, where $1 < k \leq n$. Assume that the IH holds for $i < k$, so MERGESORT correctly returns a sorted array when called on arrays of length less than k. We want to show that the IH holds for $i = k$, i.e. that MERGESORT returns a sorted array when called on an array of length k.

[INSERT INDUCTION PROOF TO PROVE THE MERGE SUBROUTINE IS CORRECT WHEN GIVEN TWO SORTED ARRAYS]

Since the two "child" recursive calls are executed on arrays of length k/2 (which is strictly less than k), then our inductive hypothesis tells us that MERGESORT will correctly sort the left and right halves of our length-k array. Then, since the MERGE subroutine is correct when given two sorted arrays, we know that MERGESORT will ultimately return a fully sorted array of length k.

Try out this inner proof on your own!

MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

In every recursive call on an array of length *at most* **i**, MERGESORT returns a sorted array.

BASE CASE

The IH holds for i = 1: A 1-element array is always sorted.

INDUCTIVE STEP *(strong/complete induction)*

Let k be an integer, where 1 < k ≤ n. Assume that the IH holds for i < k, so MERGESORT correctly returns a sorted array when called on arrays of length less than k. We want to show that the IH holds for i = k, i.e. that MERGESORT returns a sorted array when called on an array of length k.

[INSERT INDUCTION PROOF TO PROVE THE MERGE SUBROUTINE IS CORRECT WHEN GIVEN TWO SORTED ARRAYS]

Since the two "child" recursive calls are executed on arrays of length k/2 (which is strictly less than k), then our inductive hypothesis tells us that MERGESORT will correctly sort the left and right halves of our length-k array. Then, since the MERGE subroutine is correct when given two sorted arrays, we know that MERGESORT will ultimately return a fully sorted array of length k.

Try out this inner proof on your own!

CONCLUSION

By induction, we conclude that the IH holds for all 1 ≤ i ≤ n. In particular, it holds for i = n, so in the top recursive call, MERGESORT returns a sorted array.

سوال؟

زمان اجرای مرتب سازی ادغامی

چقدر سریع است؟

MERGESORT: IS IT FAST?

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

CLAIM: MergeSort runs in time **O(n log n)**

AN ASIDE: $O(n \log n)$ vs. $O(n^2)$?

log(n) grows very slowly! (Much more slowly than n)

AN ASIDE: O(n log n) vs. O(n^2)?

log(n) grows very slowly! (Much more slowly than n)

ALL LOGARITHMS IN THIS COURSE ARE BASE 2

log(2) = 1

log(4) = 2

…

log(64) = 6

log (128) = 7

…

log(4096) = 12

…

log(**# particles in the universe**) < 280

AN ASIDE: $O(n \log n)$ vs. $O(n^2)$?

log(n) grows very slowly! (Much more slowly than n)

ALL LOGARITHMS IN THIS COURSE ARE BASE 2

log(2) = 1

log(4) = 2

…

log(64) = 6

log (128) = 7

…

log(4096) = 12

…

log(**# particles in the universe**) < 280

n log n grows much more slowly than n^2

Punchline: A running time of $O(n \log n)$ is a LOT better than $O(n^2)$

MERGESORT: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:
THE RECURSION TREE!
(and we'll add up all the work done across levels to compute the Big-O runtime)

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

MERGESORT: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:

THE RECURSION TREE!

(and we'll add up all the work done across levels to compute the Big-O runtime)

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

n iterations, O(1) work per iteration

We can see that MERGE is **O(n)**

MERGESORT: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:
THE RECURSION TREE!
(and we'll add up all the work done across levels to compute the Big-O runtime)

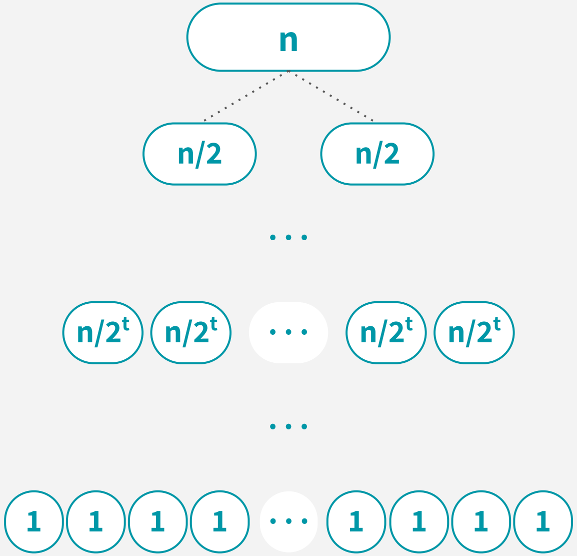
```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE(L,R):
    result = length n array
    ...
    return result
```

This means that within one recursive call that processes an array/subarray of length ***n***, the work done in that subproblem (creating subproblems & "merging" those results) is **O(n)**.

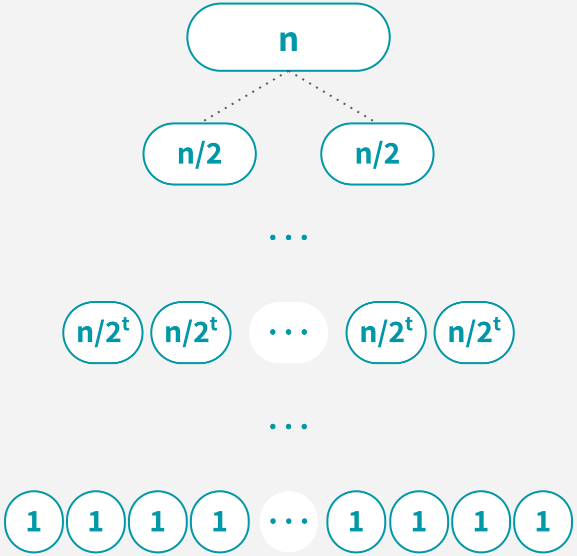
We can see that MERGE is **O(n)**

MERGESORT RECURSION TREE



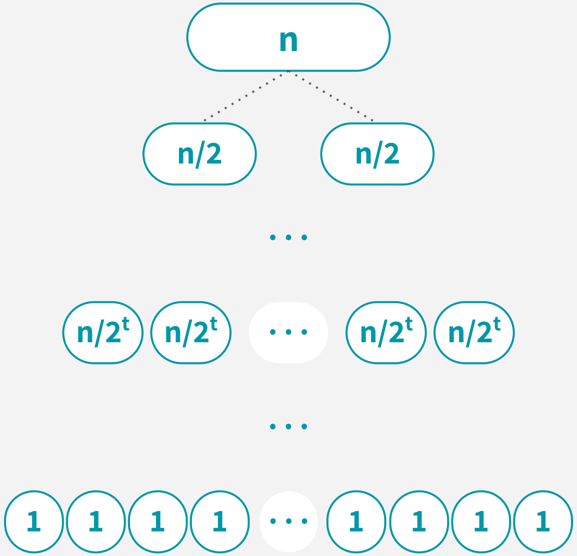
Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0				
1				
...				
t				
...				
?				

MERGESORT RECURSION TREE



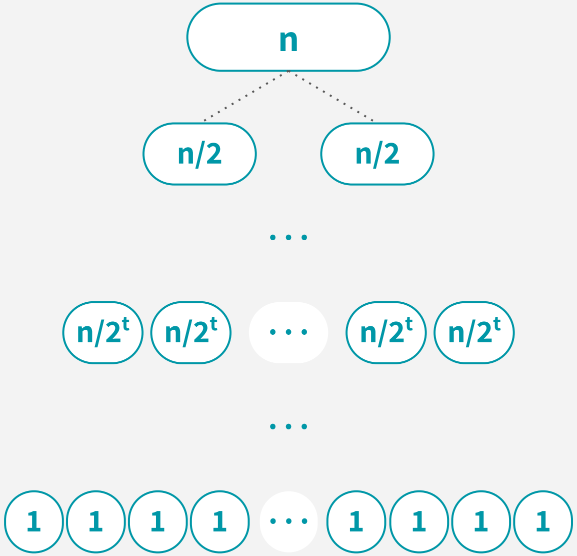
Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0				
1				
...				
t				
...				
$\log_2 n$				

MERGESORT RECURSION TREE



Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n			
1	$n/2^1$			
…				
t	$n/2^t$			
…				
$\log_2 n$	1			

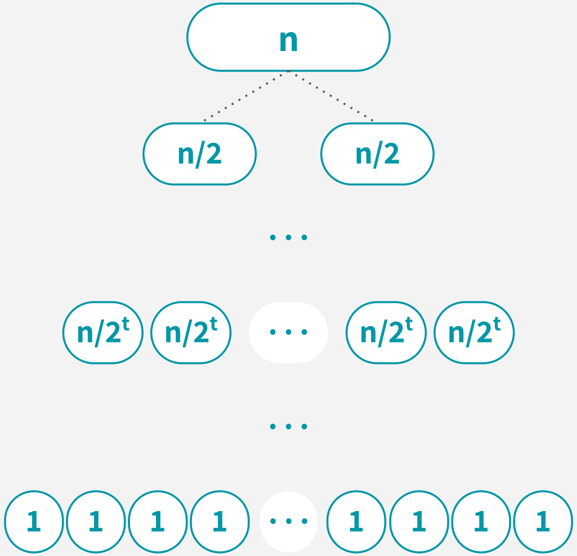
MERGESORT RECURSION TREE



Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1		
1	$n/2^1$	2^1		
		...		
t	$n/2^t$	2^t		
		...		
$\log_2 n$	1	$2^{\log_2 n} = n$		

MERGESORT RECURSION TREE

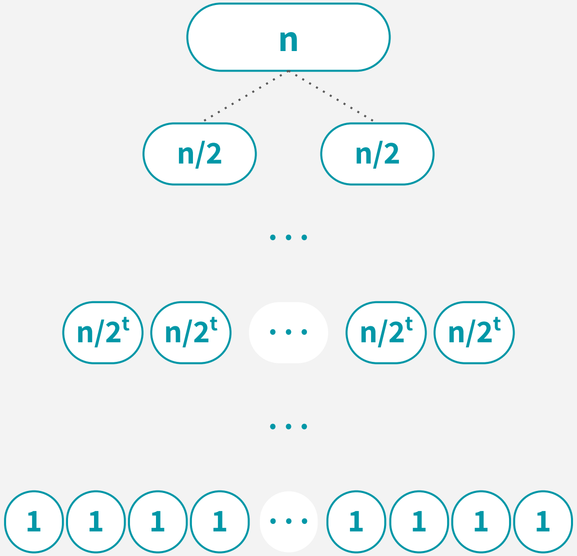
If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)



Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1	$c \cdot n$	
1	$n/2^1$	2^1	$c \cdot (n/2)$	
…				
t	$n/2^t$	2^t	$c \cdot (n/2^t)$	
…				
$\log_2 n$	1	$2^{\log_2 n} = n$	$c \cdot (1)$	

MERGESORT RECURSION TREE

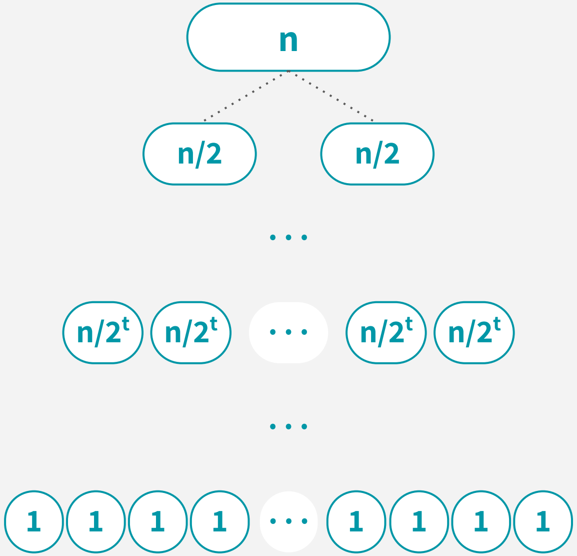
If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)



Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1	$c \cdot n$	**O(n)**
1	$n/2^1$	2^1	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) =$ **O(n)**
…				
t	$n/2^t$	2^t	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) =$ **O(n)**
…				
$\log_2 n$	1	$2^{\log_2 n} = n$	$c \cdot (1)$	$n \cdot c \cdot (1) =$ **O(n)**

MERGESORT RECURSION TREE

If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)

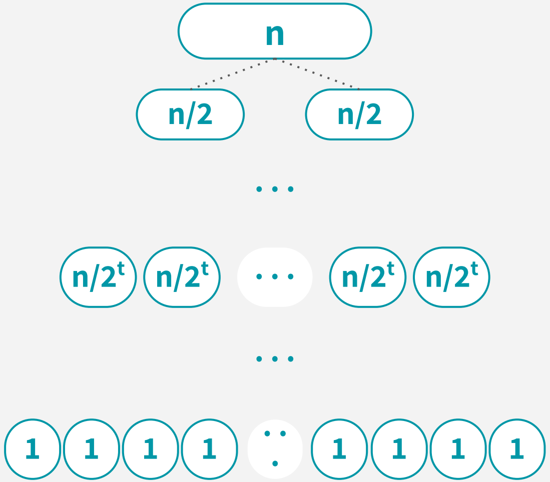


Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1	$c \cdot n$	**O(n)**
1	$n/2^1$	2^1	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) =$ **O(n)**
		. . .		
t	$n/2^t$	2^t	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) =$ **O(n)**
		. . .		
$\log_2 n$	1	$2^{\log_2 n} = n$	$c \cdot (1)$	$n \cdot c \cdot (1) =$ **O(n)**

We have ($\log_2 n + 1$) levels, each level has O(n) work total ⇒ **O(n log n)** work overall!

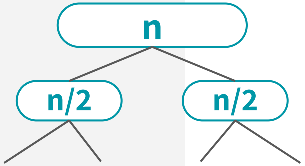
MERGESORT: O(n log n) RUNTIME

Using the "Recursion Tree Method" (i.e. drawing the tree & filling out the table),
we showed that the runtime of MergeSort is **O(n log n)**



Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1	$c \cdot n$	**O(n)**
1	$n/2^1$	2^1	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) =$ **O(n)**
		...		
t	$n/2^t$	2^t	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) =$ **O(n)**
		...		
$\log_2 n$	1	$2^{\log_2 n} = n$	$c \cdot (1)$	$n \cdot c \cdot (1) =$ **O(n)**

MERGESORT RECURRENCE RELATIONS



MergeSort

$$T(n) = 2 \cdot T(n/2) + O(n)$$

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

We can see that MERGE is **O(n)**

SOLVE WITH MASTER THEOREM

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > \\ b^d & \end{cases}$$

a: # of subproblems (branching factor)
b: factor by which input size shrinks (shrinking factor)
d: need to do $O(n^d)$ work to create subproblems + "merge" solutions

MERGESORT	**$T(n) = 2 \cdot T(n/2) + O(n)$** $T(n) =$ **$O(n \log n)$**	**$a = 2$** **$b = 2$** **$d = 1$**	**$a = b^d$**

سوال؟