KANAZAWA UNIVERSITY | Emerging Media Initiative

GSCI1801A

Information Science

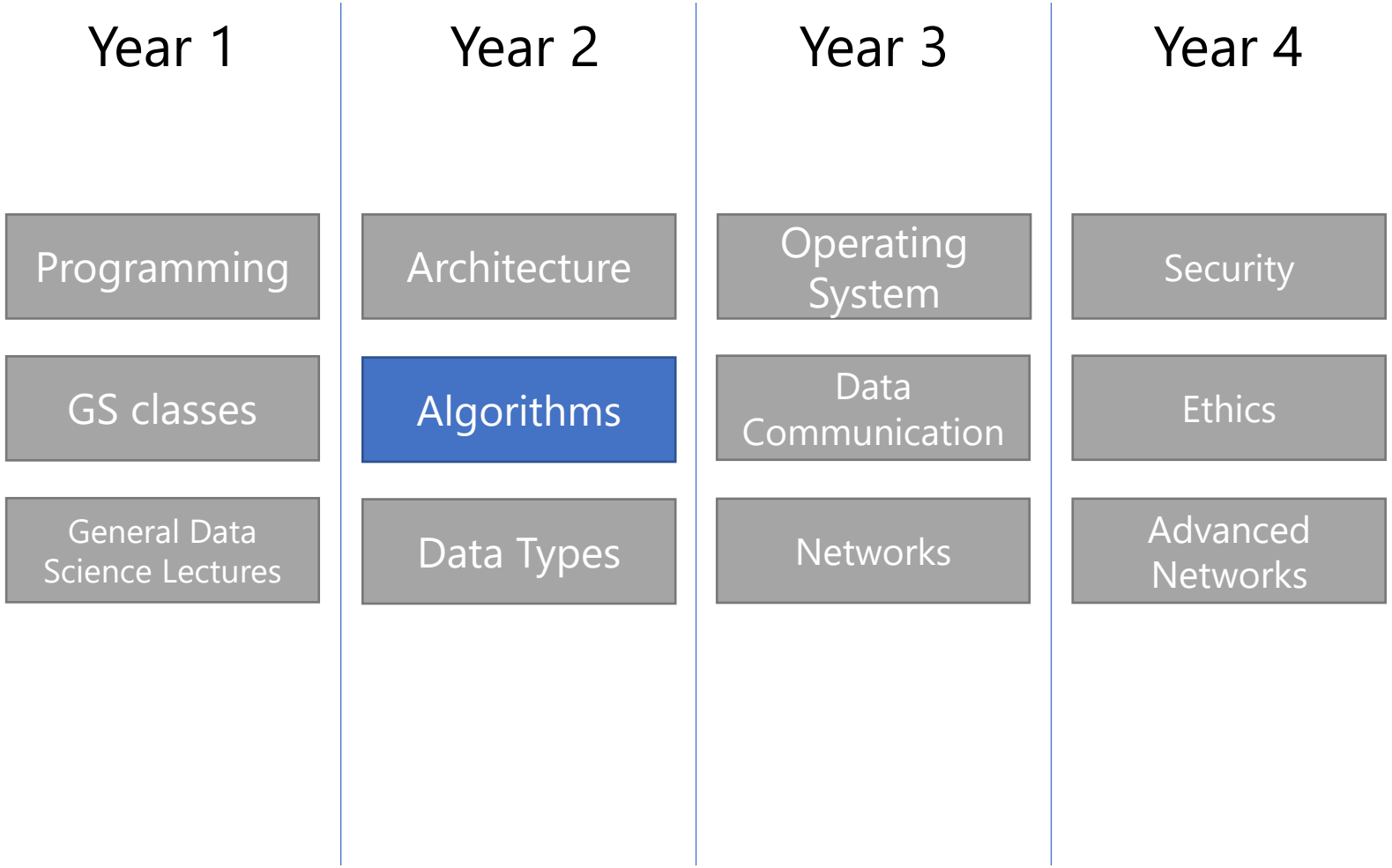
Lecture 3: Algorithms (Part 2) and Problem Solving

Asst. Prof. Chawanat NAKASAN | 2021-10-19

Agenda

1. Flowcharts & Flow Control
2. Data & Data Types
3. Algorithms
   - Time Complexity of Algorithms (Cont.)
   - Types of Algorithms
   - Examples of Algorithms
4. Problem Solving

Where are we in the CS curriculum?



This is from CS curricula in general, not specific to Kanazawa University.

Time Complexity Analysis

Importance: An algorithm can be implemented quickly or slowly!

Model example:
Prime Number Determination (IsPrime)

"Is this number a prime number?"

IsPrime

- A prime number is a positive integer ($n > 1$) where only 1 and itself (n) can divide it without remainders.

$$\forall\, i \in \mathbb{N} \cap (1, n);\ n \bmod i \neq 0$$

- Is 91 a prime number? No.
- Is 311 a prime number? Hmm.

Example of mod (modulo):

10 / 3 = 3 R 1

∴ 10 mod 3 = 1

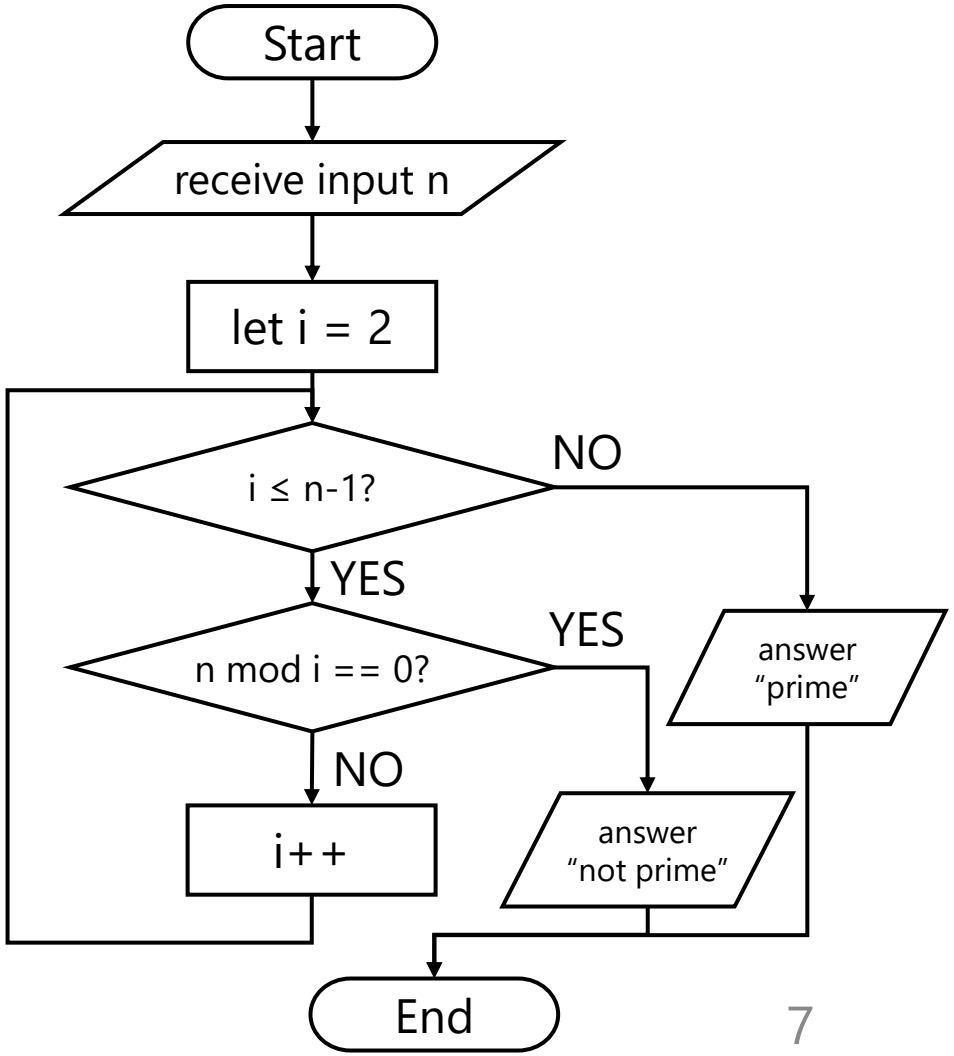
20 / 4 = 5 R 0

∴ 20 mod 4 = 0

IsPrime method 1: Naïve Brute-Forcing Every Number

- Test every divisor from 2 to n.
  - 311 mod 2 = ?
  - 311 mod 3 = ?
  - 311 mod 4 = ?
  - …
  - 311 mod 310 = ?

- You must perform at least **n-2** calculations.
- This means the time complexity of this method is **O(n)**.



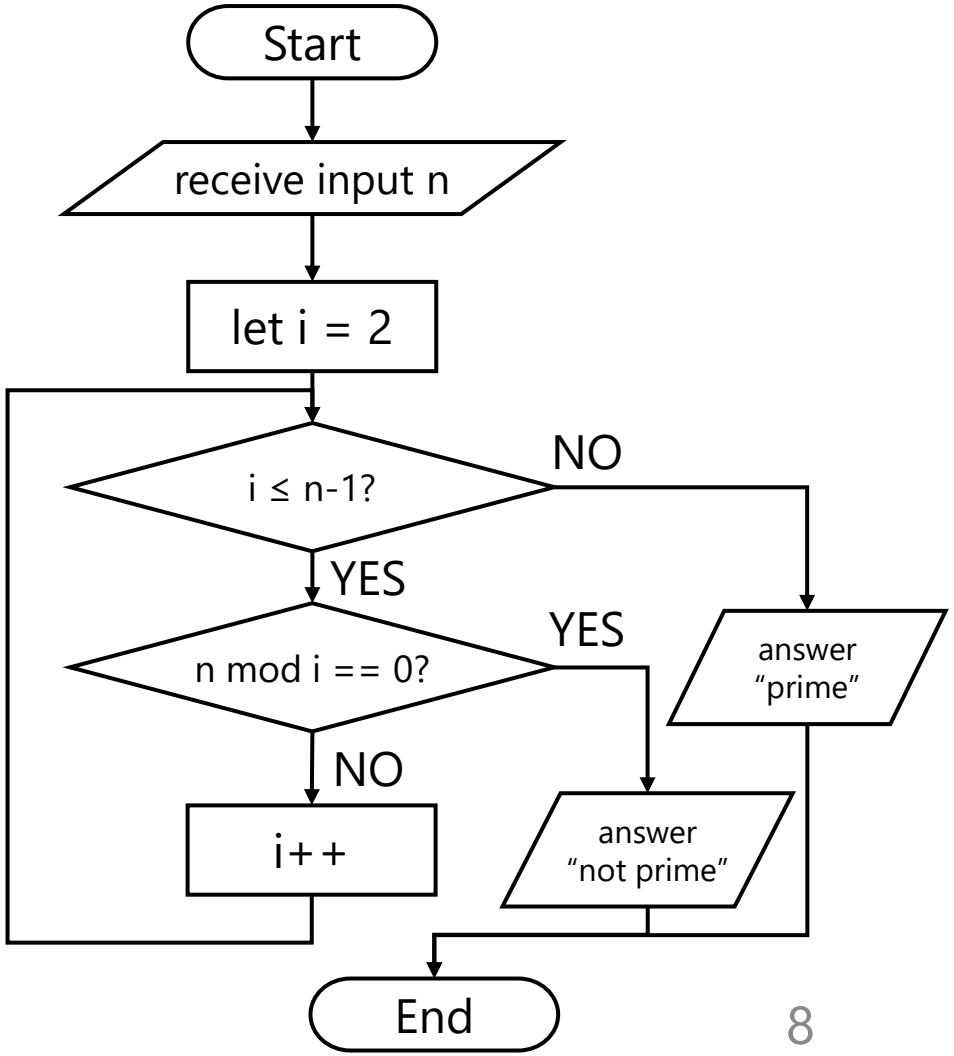
IsPrime method 1: Naïve Brute-Forcing Every Number

Flowcharts are getting a little unwieldy at this point, so I'll start writing algorithm in text instead.

Input: integer n where n > 1

1. for i ← 2…(n-1):
2. if n mod i = 0:
3. answer "not prime"
4. end if
5. i++
6. end for
7. answer "prime"

This algorithm in text is equivalent to the flowchart on the right.



IsPrime method 1: Naïve Brute-Forcing Every Number

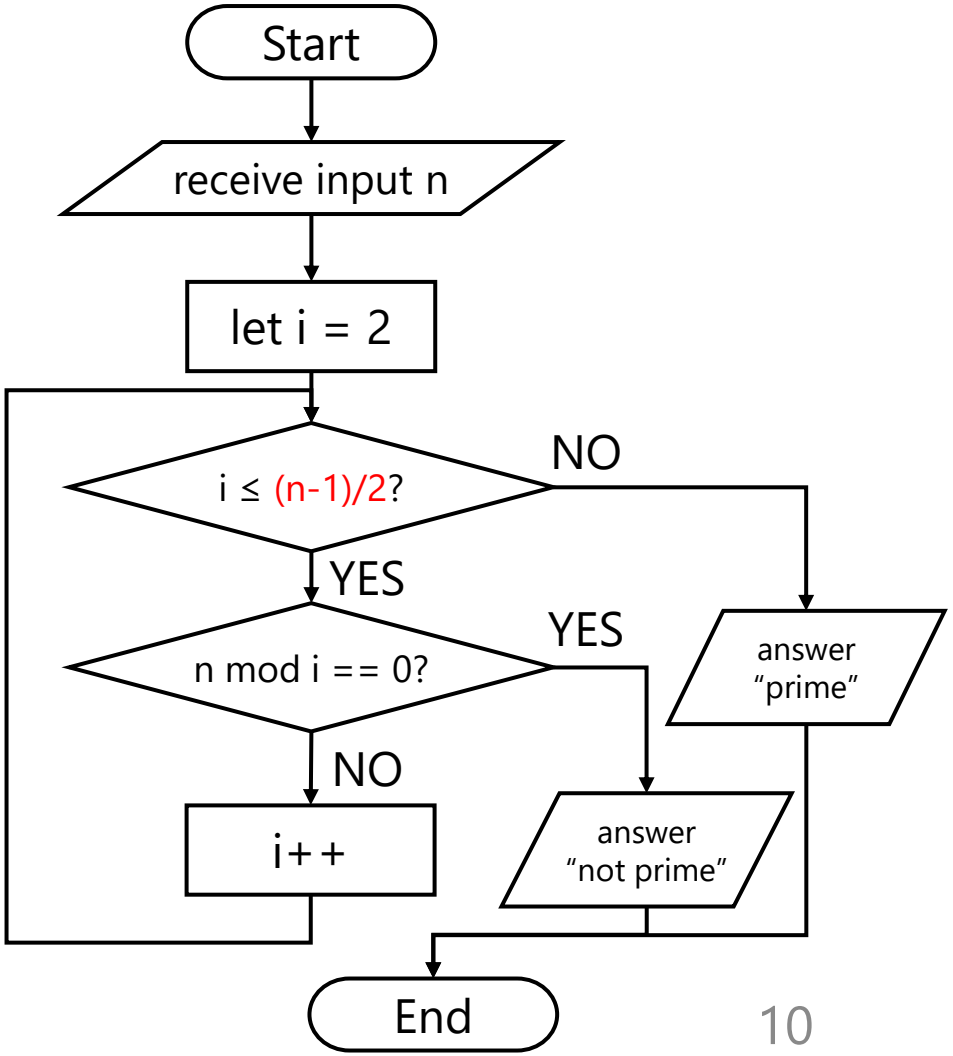
- This method is hideously slow because you're doing repeated meaningless calculations.

- For example, trying to calculate 311 mod 200 is worthless. You probably shouldn't need to try divisors larger than 105 ($\left\lfloor \frac{311}{2} \right\rfloor$), right?

IsPrime method 1.2: Naïve Brute-Forcing ~~Every~~ half of the range Number

Input: integer n where n > 1

1. for i ← 2…((n-1)/2):
2. if n mod i = 0:
3. answer "not prime"
4. end if
5. i++
6. end for
7. answer "prime"



IsPrime method 1.2: Naïve Brute-Forcing ~~Every~~ half of the range Number

Input: integer n where n > 1

1. for i ← 2…((n-1)/2):
2. if n mod i = 0:
3. answer “not prime”
4. end if
5. i++
6. end for
7. answer “prime”

- You still must perform about (n-1)/2 operations.
- Time complexity remains **O(n)**.

IsPrime method 2: Brute-Force from 2 to sqrt(n)

Input: integer n where n > 1

1. for i ← 2…sqrt(n):
2. if n mod i = 0:
3. answer "not prime"
4. end if
5. i++
6. end for
7. answer "prime"

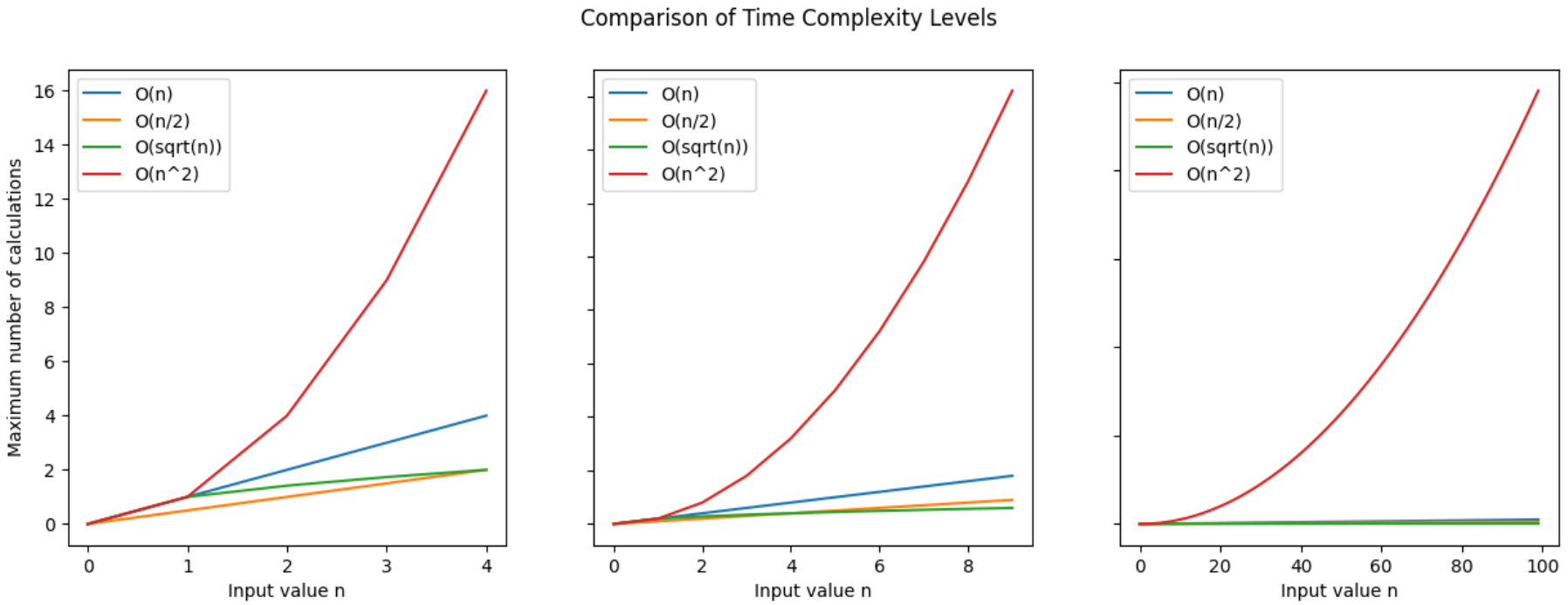
- What about this idea: Since numbers like 36 factorize into 6 x 6 for example, why don't we run from 2 to sqrt(n)?
- Time complexity now becomes O(sqrt(n)) or O($\sqrt{n}$). This is quite a progress!

Why is $O(\sqrt{n}) < O(n/2) = O(n) < O(n^2)$

Knowing a bit about number factors!

Linear Brute-Force

Remember TableCopy from last week?



Conclusion

- We have learned about the nature of some simple time complexity levels.
- We will continue to use time complexity levels in our lectures.
- Efficient algorithm design helps improve performance and conserve computation power.

Types and Examples of Algorithms

Notes

- This is not meant to be an exhaustive list of all possible algorithms.
- I'll list only algorithms that have nice and useful philosophy which can be adapted for near-future problem solving, in this class or in your life.
- You are not limited to using these types of algorithms to solve real-world problems.

Recursion Algorithms

- Recursion in algorithms is the definition of problem in a way that refers to the smaller version of the problem.

Why is recursion important?

- Many real-world problems can be solved by solving a smaller version of the problem, then doing a little extra work.
- Many real-world phenomena exhibit some degree of recursion.

Recursion: Factorial

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & otherwise \end{cases}$$

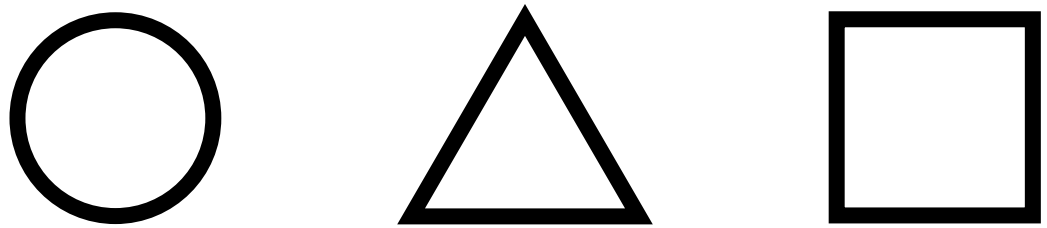
Recursion: Factorial

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1)! & otherwise \end{cases}$$

Example:

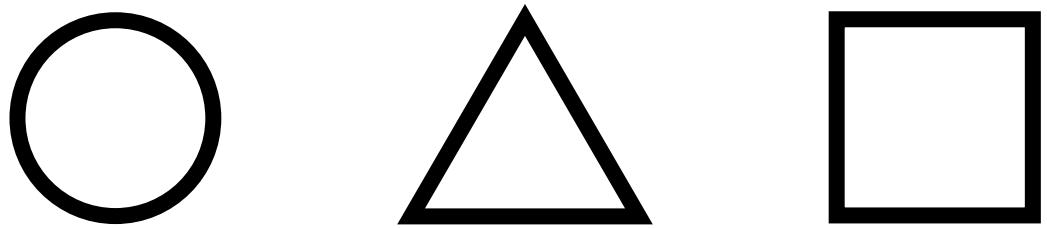
$$\begin{aligned} 3! &= 3 \cdot (3-1)! \\ &= 3 \cdot 2! \\ &= 3 \cdot 2 \cdot (2-1)! \\ &= 3 \cdot 2 \cdot 1! \\ &= 3 \cdot 2 \cdot 1 \cdot (1-1)! \\ &= 3 \cdot 2 \cdot 1 \cdot 0! \\ &= 3 \cdot 2 \cdot 1 \cdot 1 \\ &= 6 \end{aligned}$$

Real-world examples of factorials



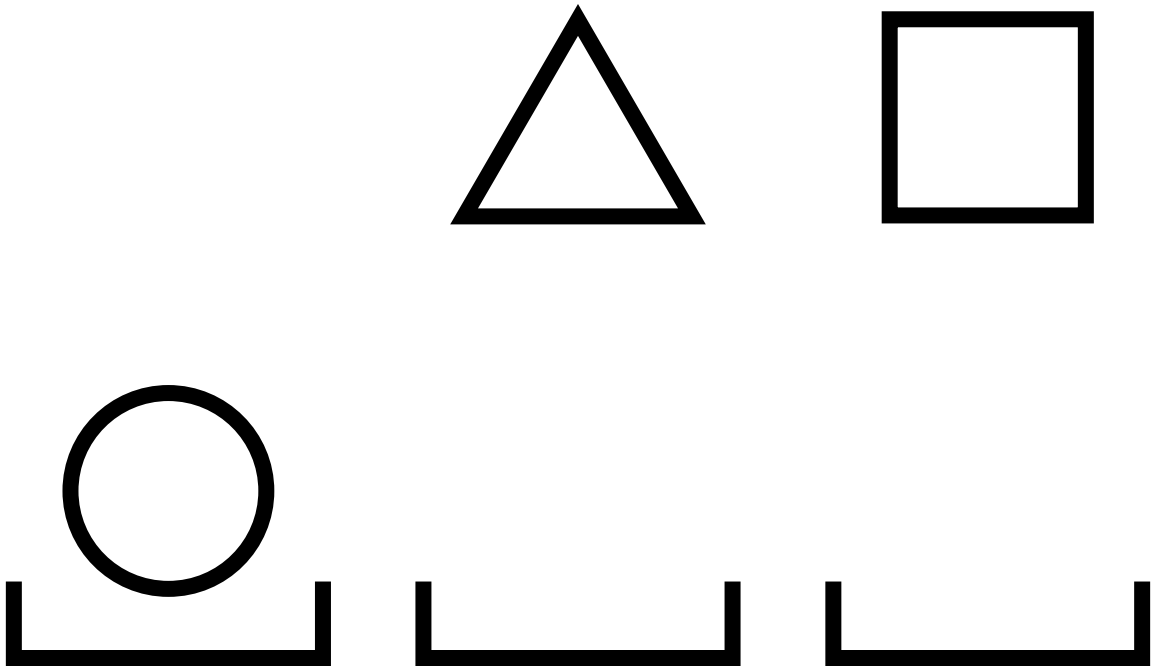
How many ways can you arrange these three shapes?

Real-world examples of factorials



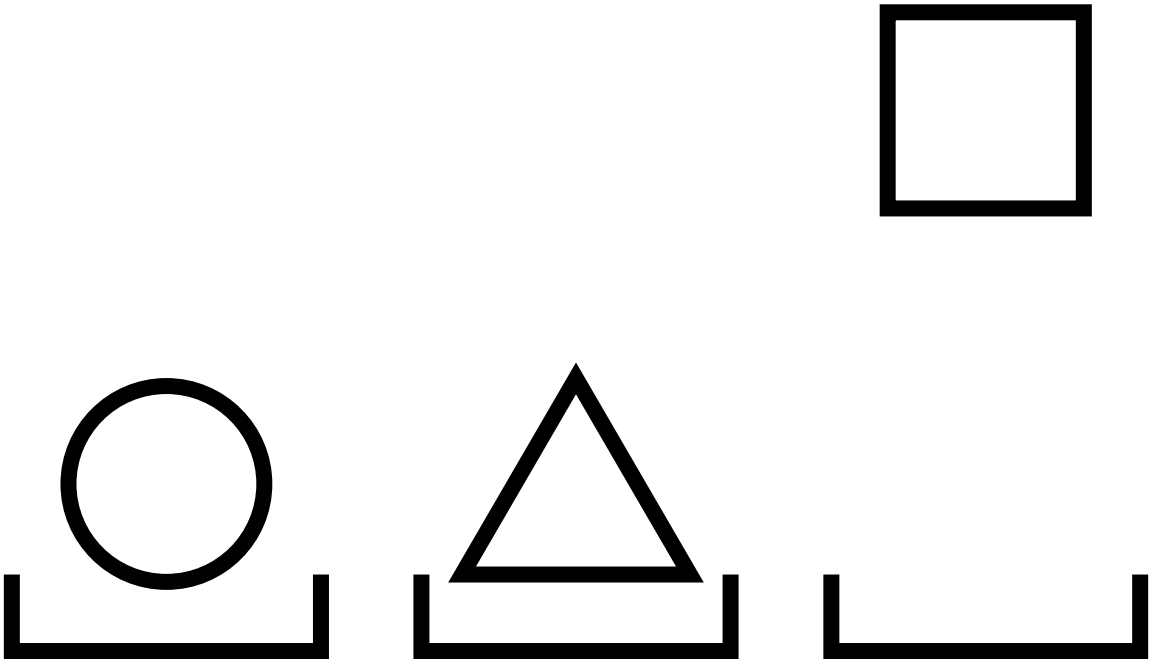
3 possible shapes

Real-world examples of factorials



3 possible shapes 2 possible shapes

Real-world examples of factorials



3 possible shapes 2 possible shapes 1 possible shape

$$3 \times 2 \times 1 = 6$$

Expand this to "n" boxes:

$$n \times (n-1) \times (n-2) \times \dots \times 1$$

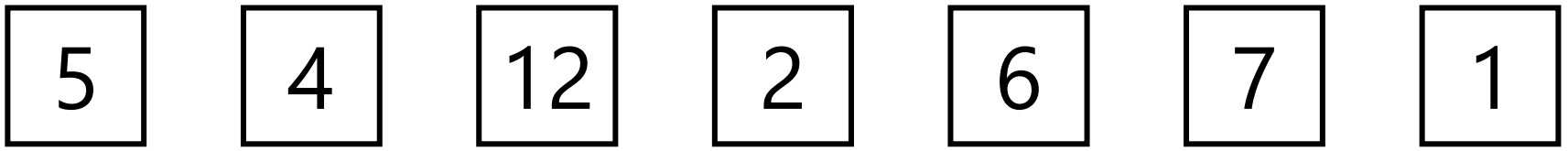
$$= \mathbf{n!}$$

This is called **permutation**.

Number of possible permutations when we have n boxes to contain n different things is n!
(There are other, more specific cases, but you will learn that in discrete mathematics.)

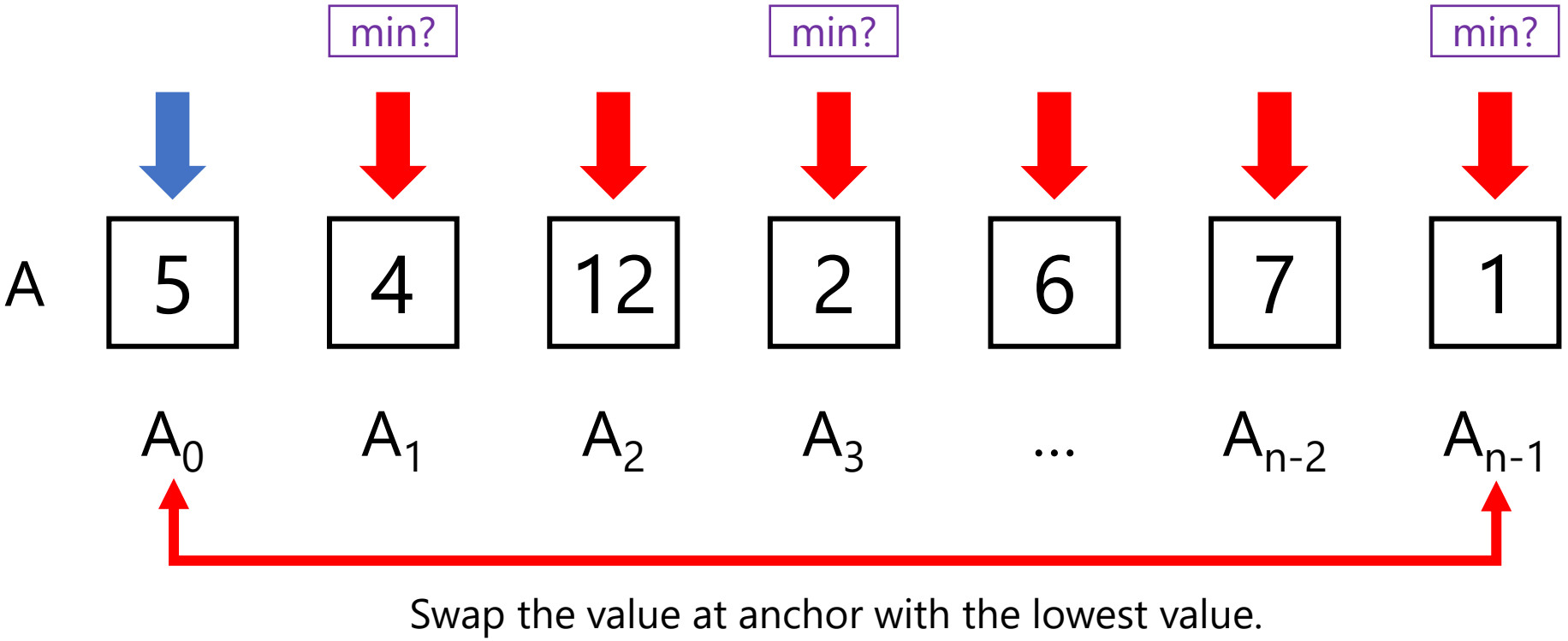
Recursion: Sorting

- Recursion is also very useful for sorting.
- How do you normally sort numbers?
- **TASK**: Sort the following number from smallest to largest.



Sorting: Selection Sort:

Let A be the data array.

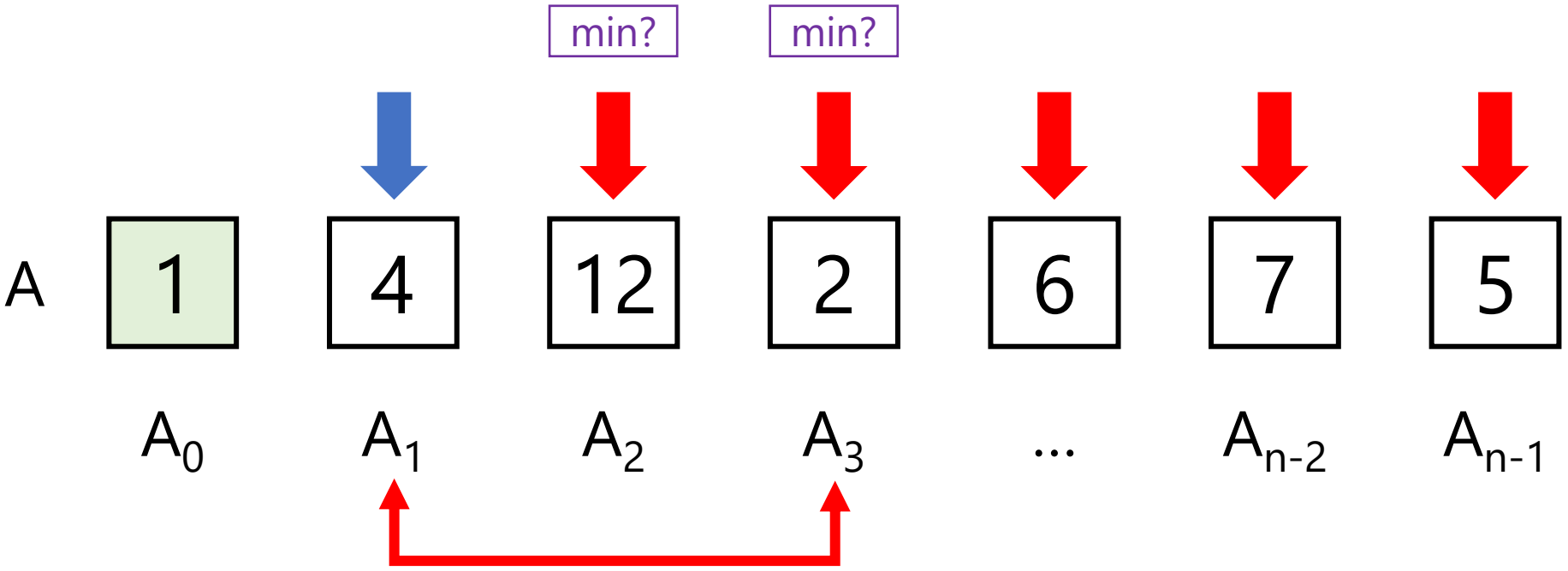


Swap the value at anchor with the lowest value.

Remember: Computers are not very smart!

Sorting: Selection Sort :

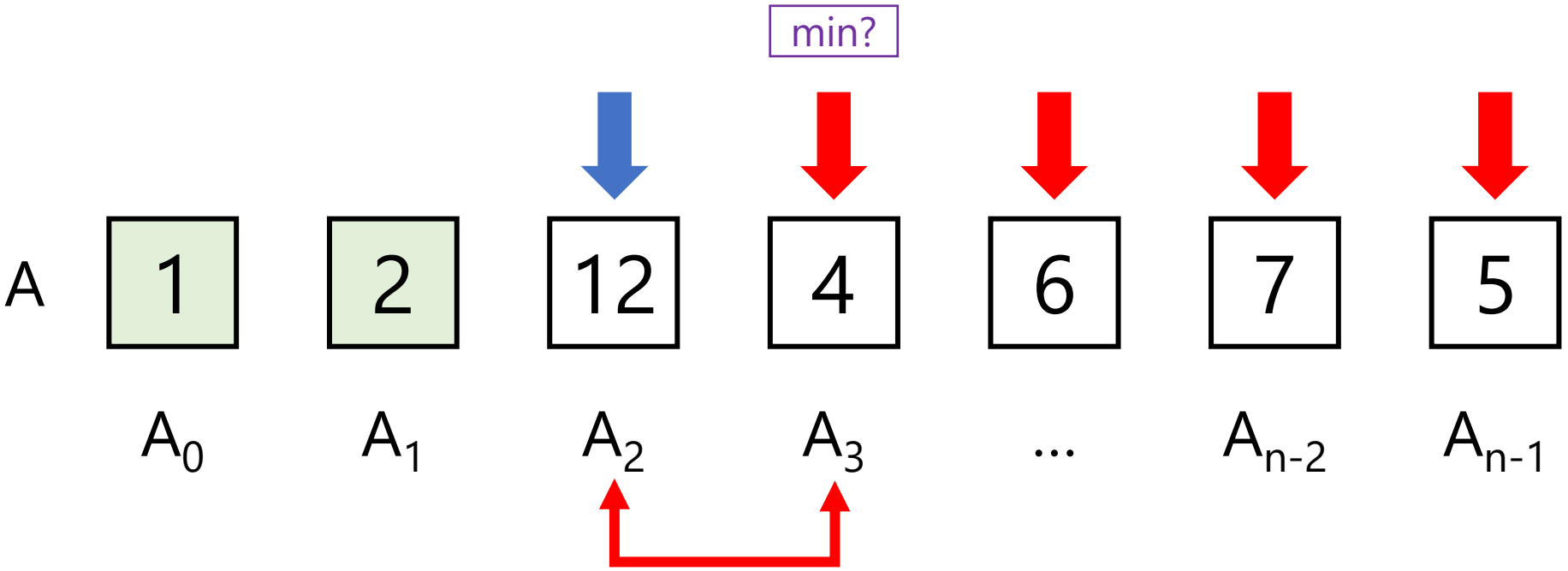
Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

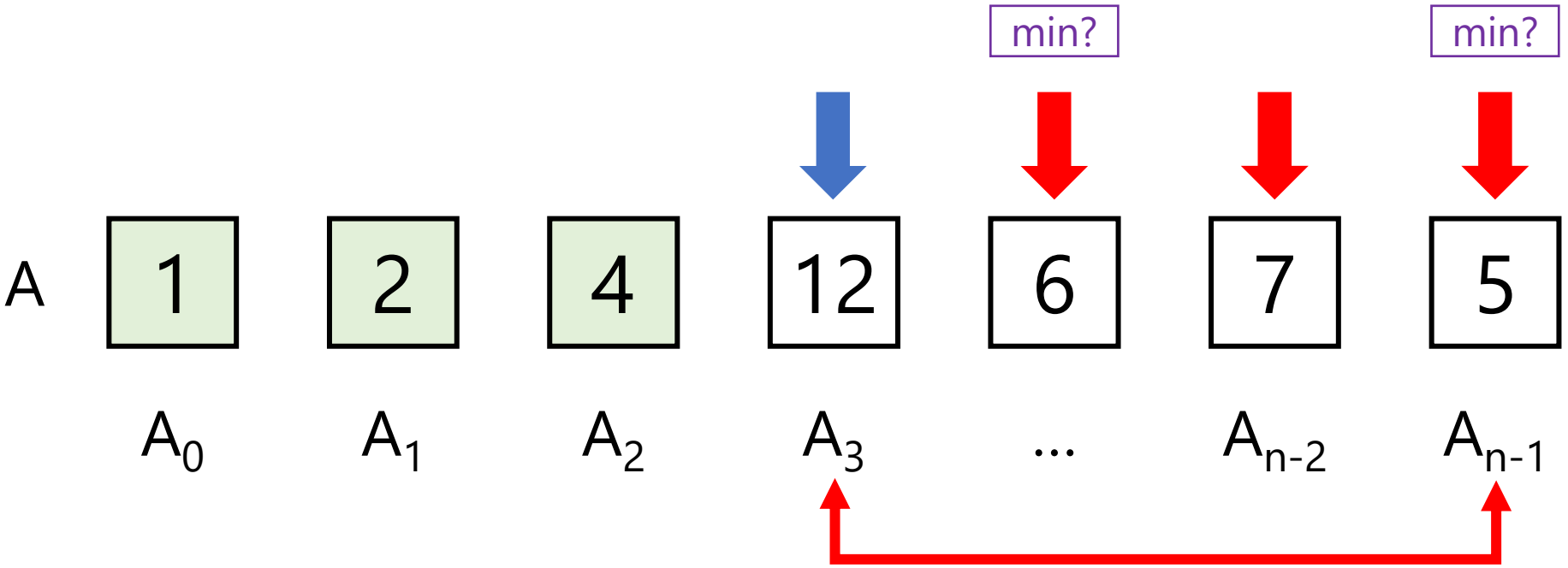
Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

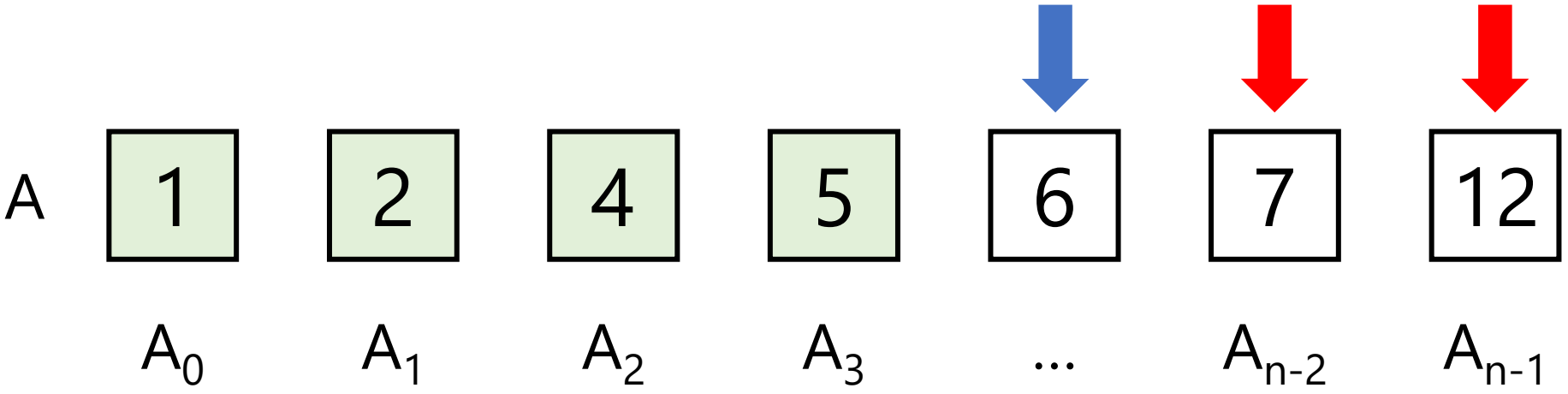
Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

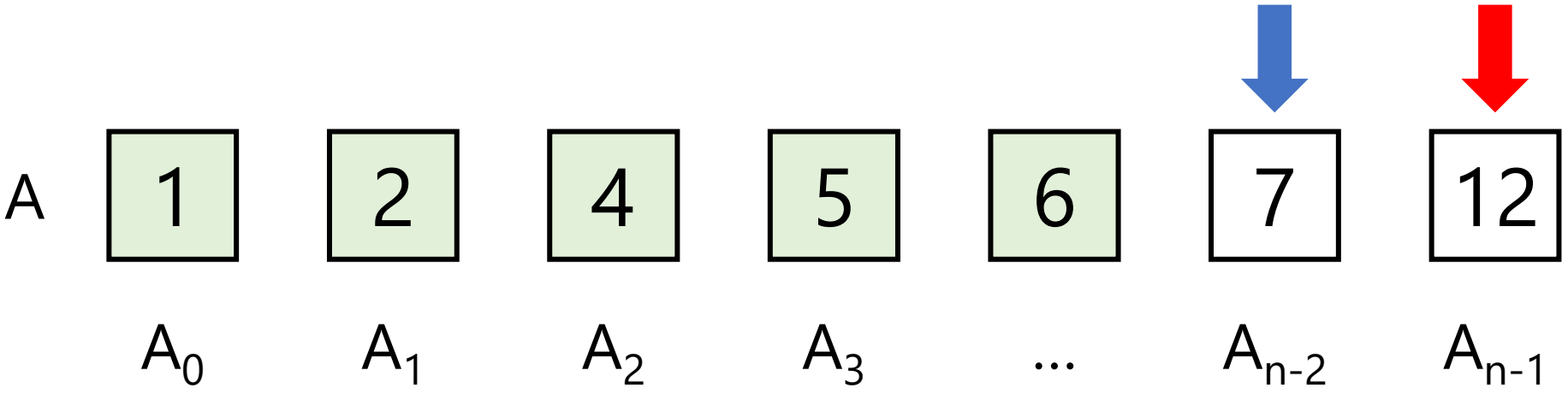
Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

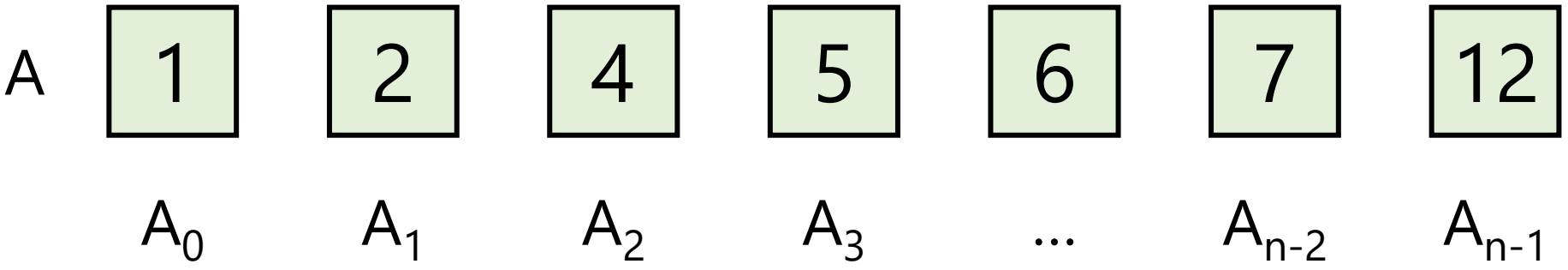
Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

Let A be the data array.



Remember: Computers are not very smart!

Sorting: Selection Sort :

- The algorithm is very simple, yes, but very cumbersome.
- For every position *i* (an O(n) task), we have to scan the whole range (except what we've already done.) Scanning the whole range is an O(n) job.
- An O(n) job stacked in another O(n) job results in **$O(n^2)$** total time complexity.

Input array A containing *n* numeric values

Outer loop (select anchor and swap)

```
1. for i ← 0 ... (n-2):
2.     let MIN ← ∞, MININDEX ← 0
3.     for j ← i+1 ... (n-1):
4.         if A_i < MIN:
5.             MIN ← A_i
6.             MININDEX ← I
7.     swap A_i, A_j
8. return A (as answer)
```

Inner loop (scan for min value)

Don't worry if you don't get it right now. When you get home, grab some playing cards and mimic what we've done.

Sorting: Quicksort: What is

- Quicksort is a recursive algorithm. It is notably much faster, and algorithmically simpler than selection sort.

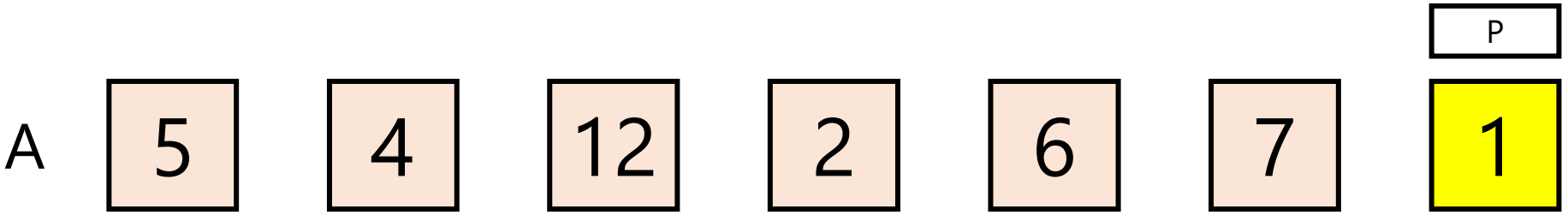
Sorting: Quicksort: How to

- To quicksort:
  - If the range to sort has 1 value, do nothing.
  - If the range to sort has 2 values, swap them if needed.
  - Otherwise:
    - Select the last element as the pivot. ※
    - Partition the array into left side (values less than pivot) and right side (values greater than pivot).
    - Move pivot in between the left and right sides.
    - Quicksort the left side of the pivot.
    - Quicksort the right side of the pivot.

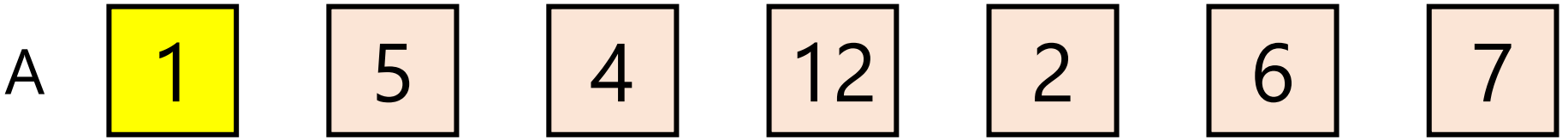
※ There are many ways to select a pivot.

Sorting: Quicksort Example (1)

Select Pivot



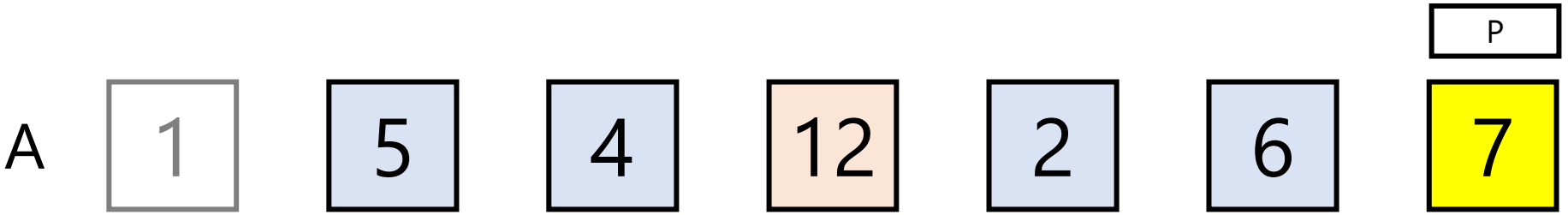
Arrange the array.



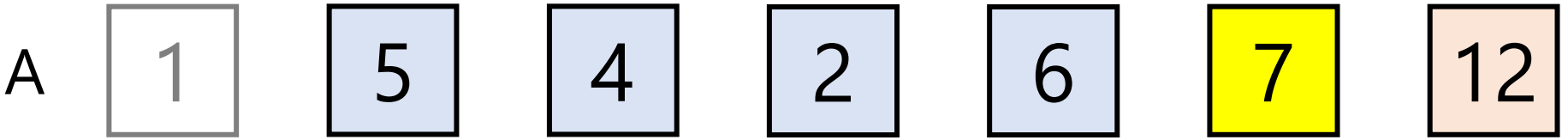
Sort the left side: there is no left side.
Sort the right side: there are 6 values, so we must go deeper.

Sorting: Quicksort Example (2)

Select Pivot



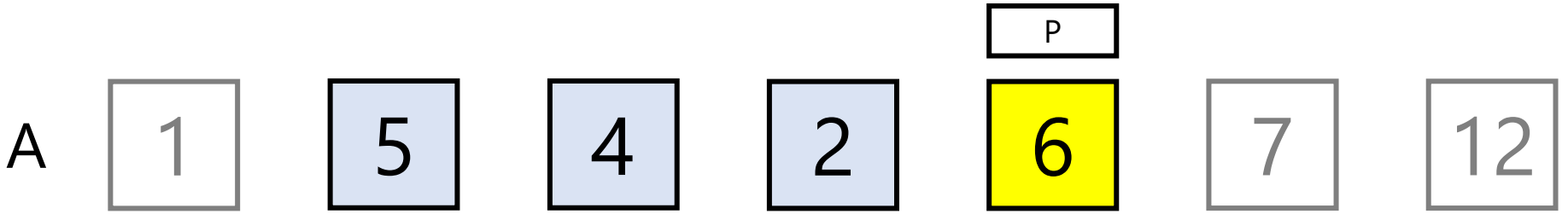
Arrange the array.



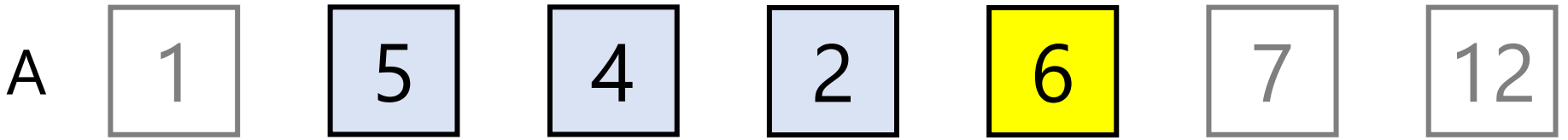
Sort the left side: there are 4 elements.
Sort the right side: right side has 1 element ➔ do nothing

Sorting: Quicksort Example (3)

Select Pivot



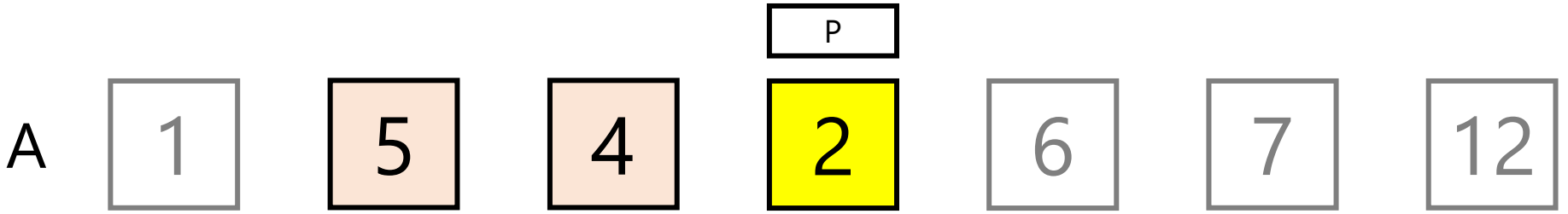
Arrange the array.



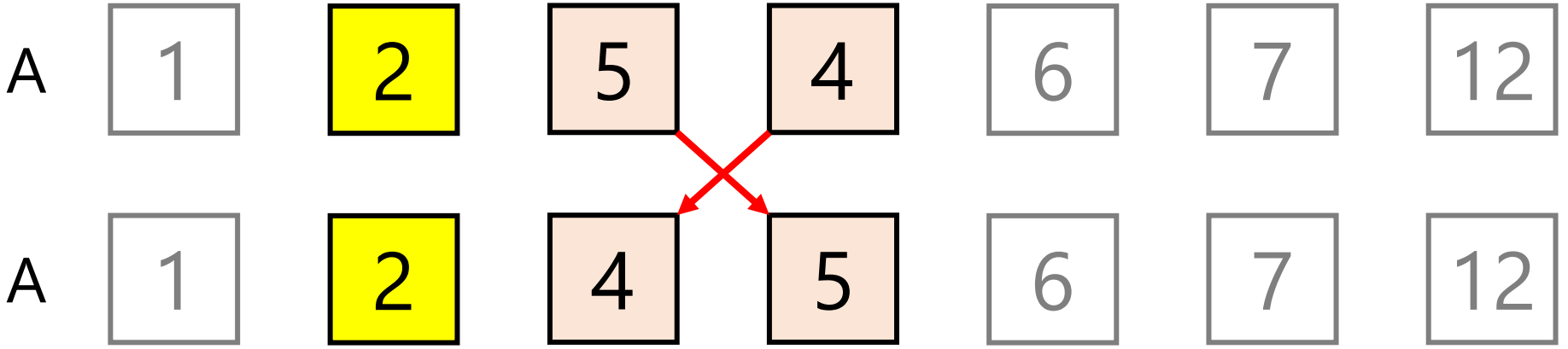
Sort the left side: there are 3 elements.
Sort the right side: right side has 0 elements → do nothing.

Sorting: Quicksort Example (4)

Select Pivot



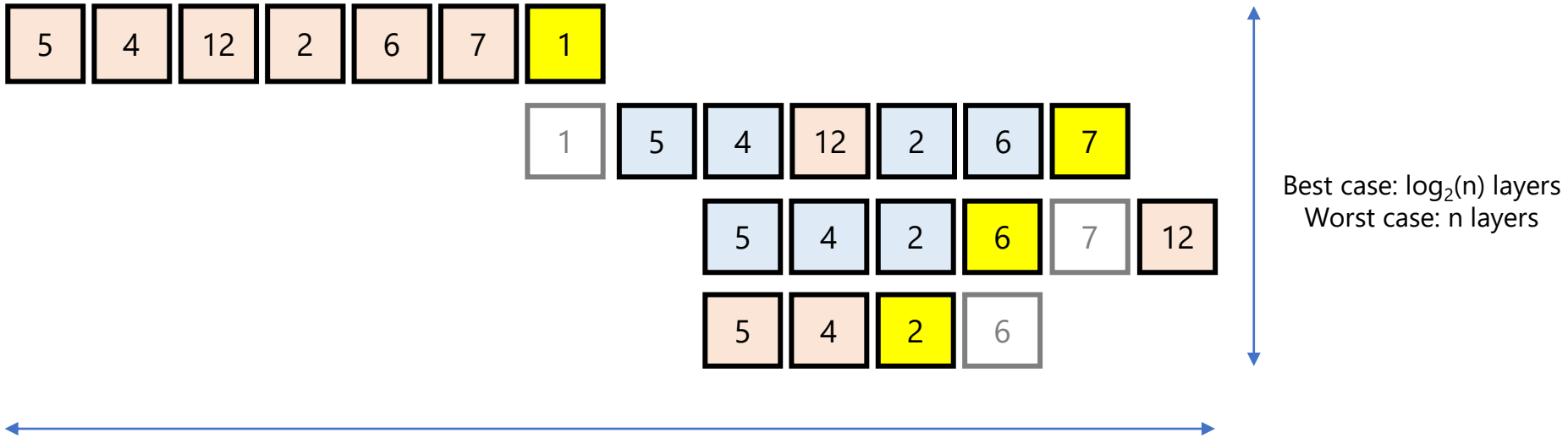
Arrange the array.



Sort the left side: there are 0 elements → do nothing.
Sort the right side: right side has 2 elements → swap if needed.

Time Complexity of Quicksort

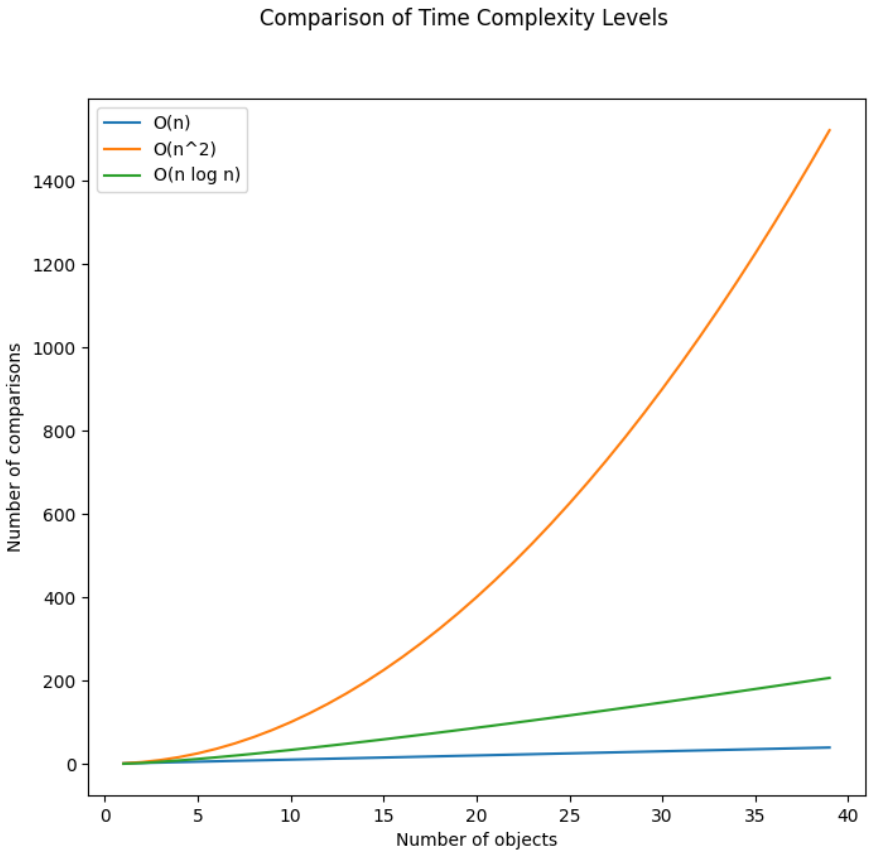
- Quicksort really depends on luck and pivot selection choice. However, no matter what happens, it at least ensures that almost all value "visits" are meaningful.



In each layer, all values are visited once → O(n)

Total time complexity for quicksort is O(n log n) best case, or $O(n^2)$ worst case.

Time Complexity of Quicksort



Average case for selection sort
Worst case for quicksort

Average case for quicksort

Advantages of Quicksort

- Very easy to write and implement compared to other O(n log n) algorithms. (This is why I chose to present Quicksort today.)
- Smaller memory footprint compared to Merge Sort (you might want to look into this if you want to do report on algorithms – hint hint!).
- Almost always faster than Selection Sort (naïve sort).

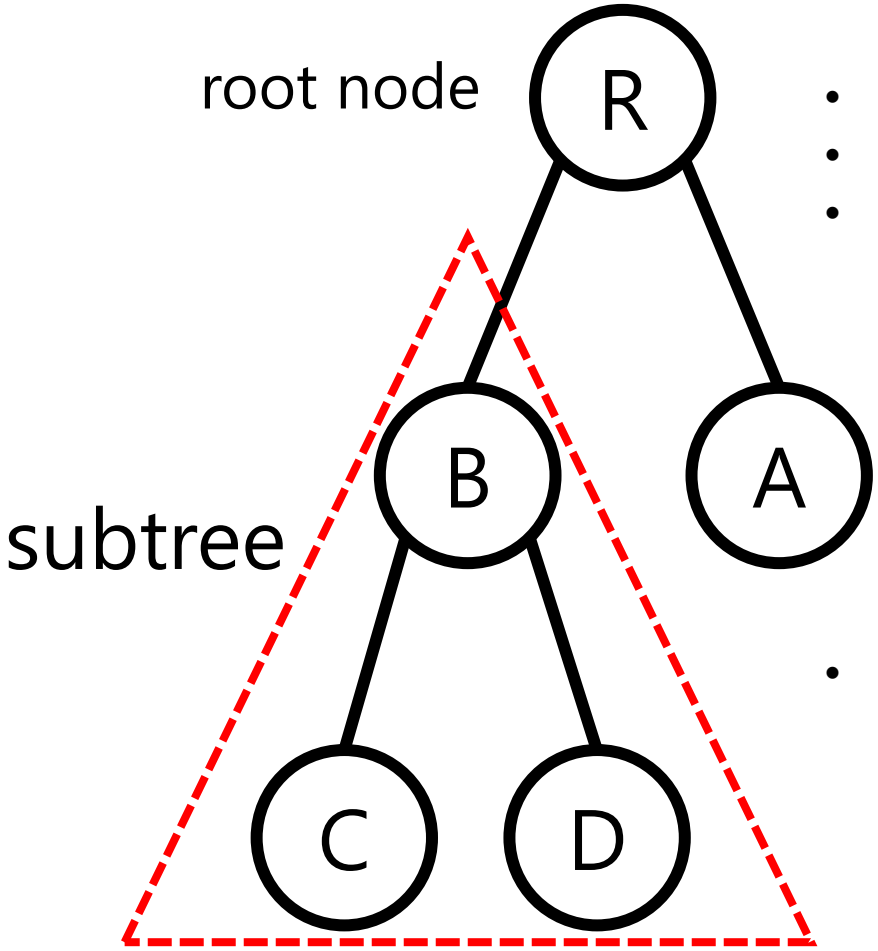
Limitations of Quicksort

- Quicksort can slow down if pivot selection is suboptimal (bad choice of pivot, or just being unlucky). *This is still usually faster than Selection Sort*.

- Recursions causes your operating system to keep track of individual function calls using the stack memory.
- This is not a problem in most modern computers, but Quicksort may run into memory trouble in microprocessor programming.

- You will learn about this further if you take a course in operating systems or system software programming.

Recursion: Tree Structure

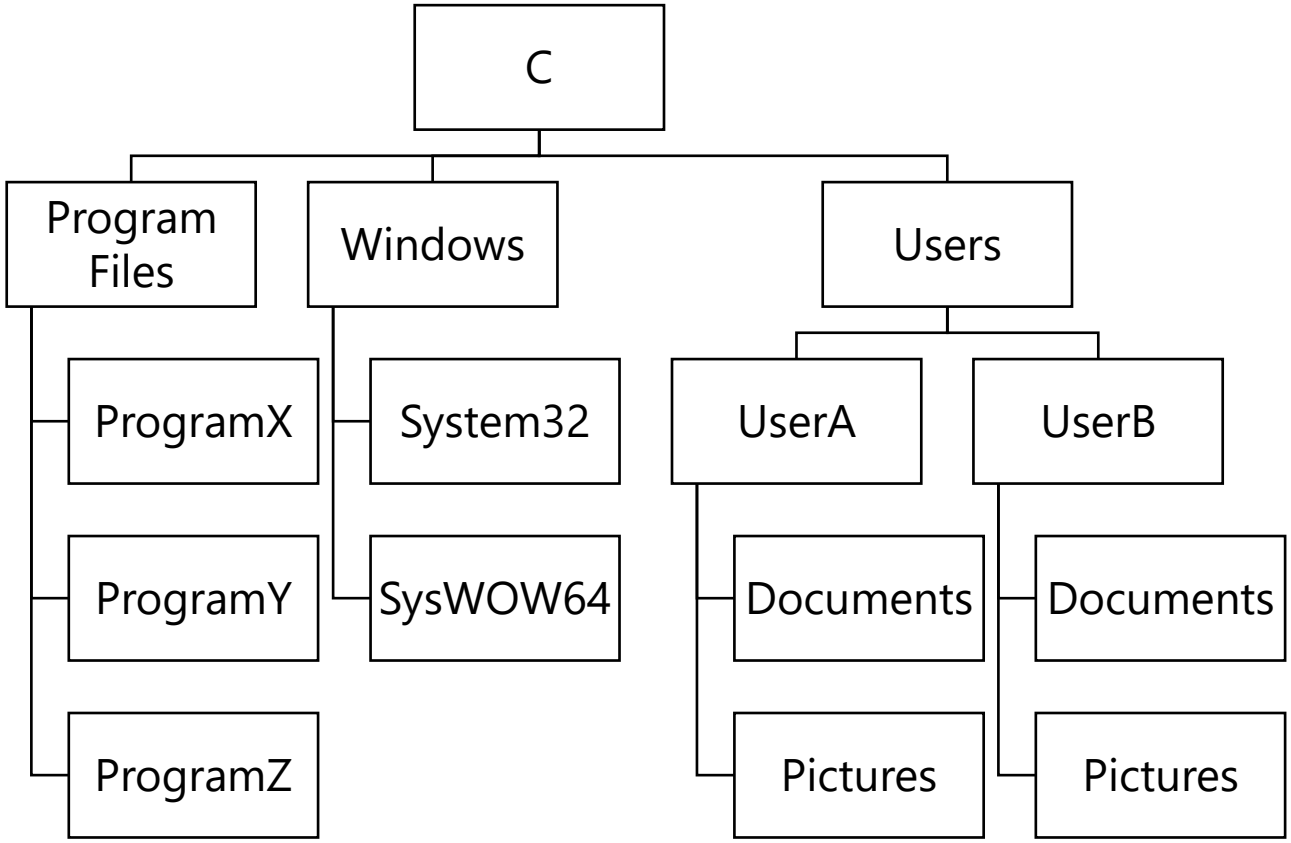


- A **tree** consists of **nodes**.
- The **root node** has no parents.
- Other nodes have one parent. For example, node R is the parent of nodes A and B.

- You can see that a hierarchy can be created, with a smaller tree inside a larger tree. For example, the B-C-D structure is itself a tree.

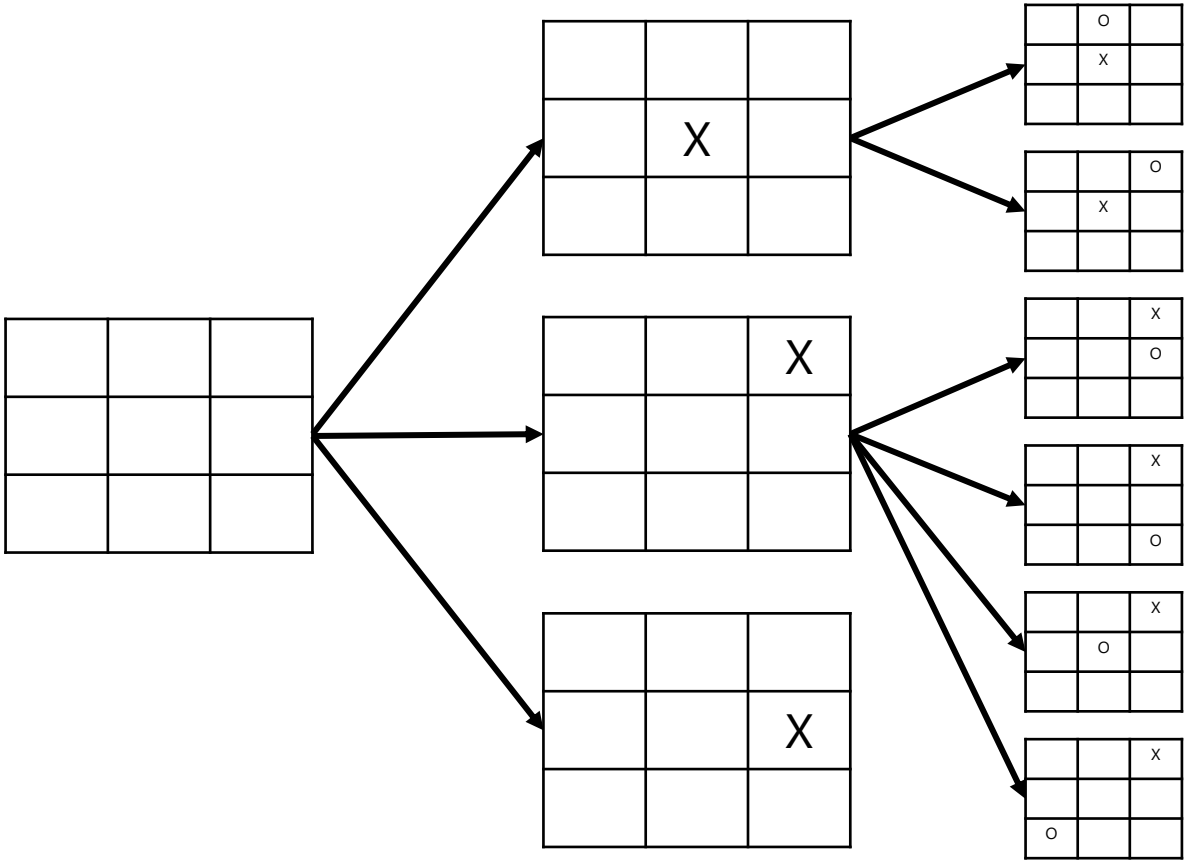
Tree structure is *very important* for:

***Anything with* Hierarchies:** file systems, etc.



Tree structure is *very important* for:

State hierarchy (board games, etc.)



Greedy Algorithms

- Greedy algorithms focus on obtaining immediately better results without focusing on long-term (global) results.
- Many times, you don't need *the best* solution. You need *good enough* solution. Greedy algorithms are great for this.

The Worship of Mammon (Evelyn De Morgan, c1909). This picture illustrates a person who, beyond simply money, worships the *idea of money* itself, leading to her own doom.

Knapsack Problem

- You have a backpack. It fits 15 kg.
- There are five objects, each with different weight (w_i) and value (v_i).
- You must fill your backpack with the greatest value possible, but without overloading yourself.
- **How do you solve this problem?**

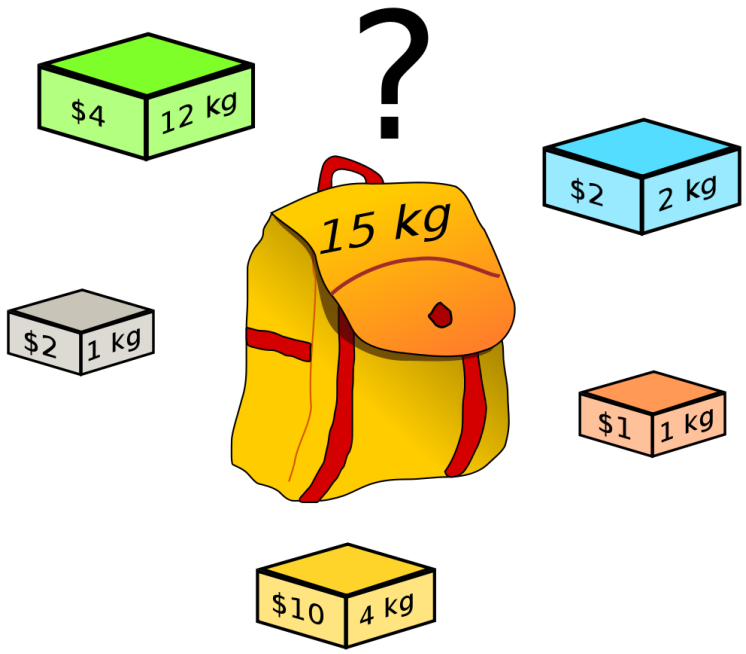
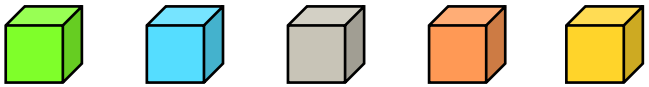


Image Source: https://en.wikipedia.org/wiki/Knapsack_problem

Knapsack Problem: Brute-Force?



$4 12kg	$2 2kg	$2 1kg	$1 1kg	$10 4kg	Weight $\sum w_i$	Value $\sum v_i$
					0	0
				✓	4	10
			✓		1	1
			✓	✓	5	11
		✓			1	2
		✓		✓	5	12
		✓	✓		2	3
		✓	✓	✓	6	13
	✓				2	2
etc.						

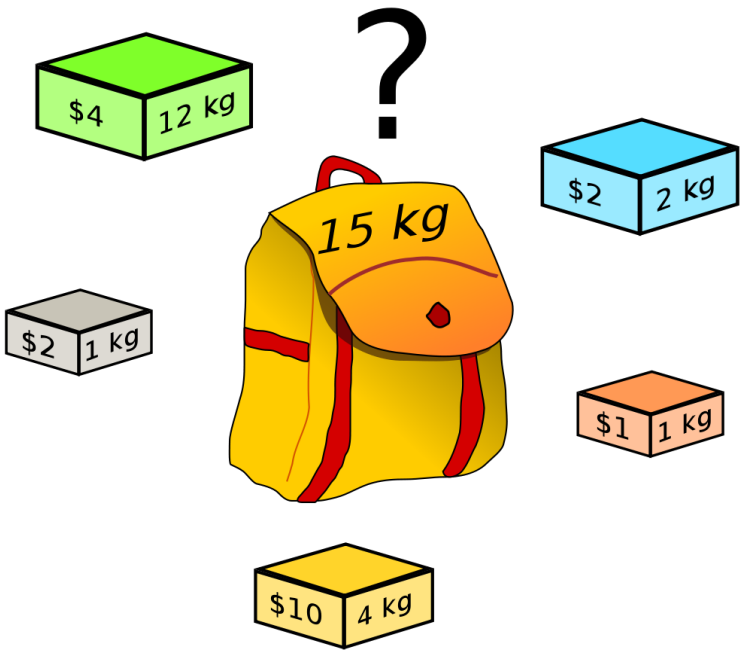
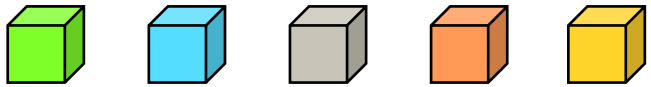


Image Source: https://en.wikipedia.org/wiki/Knapsack_problem

Knapsack Problem: Brute-Force?



$4 12kg	$2 2kg	$2 1kg	$1 1kg	$10 4kg	Weight $\sum w_i$	Value $\sum v_i$
					0	0
				✓	4	10
			✓		1	1
			✓	✓	5	11
		✓			1	2
		✓		✓	5	12
		✓	✓		2	3
		✓	✓	✓	6	13
	✓				2	2
etc.						

Image Source: https://en.wikipedia.org/wiki/Knapsack_problem

The problem of this method:

Time complexity.

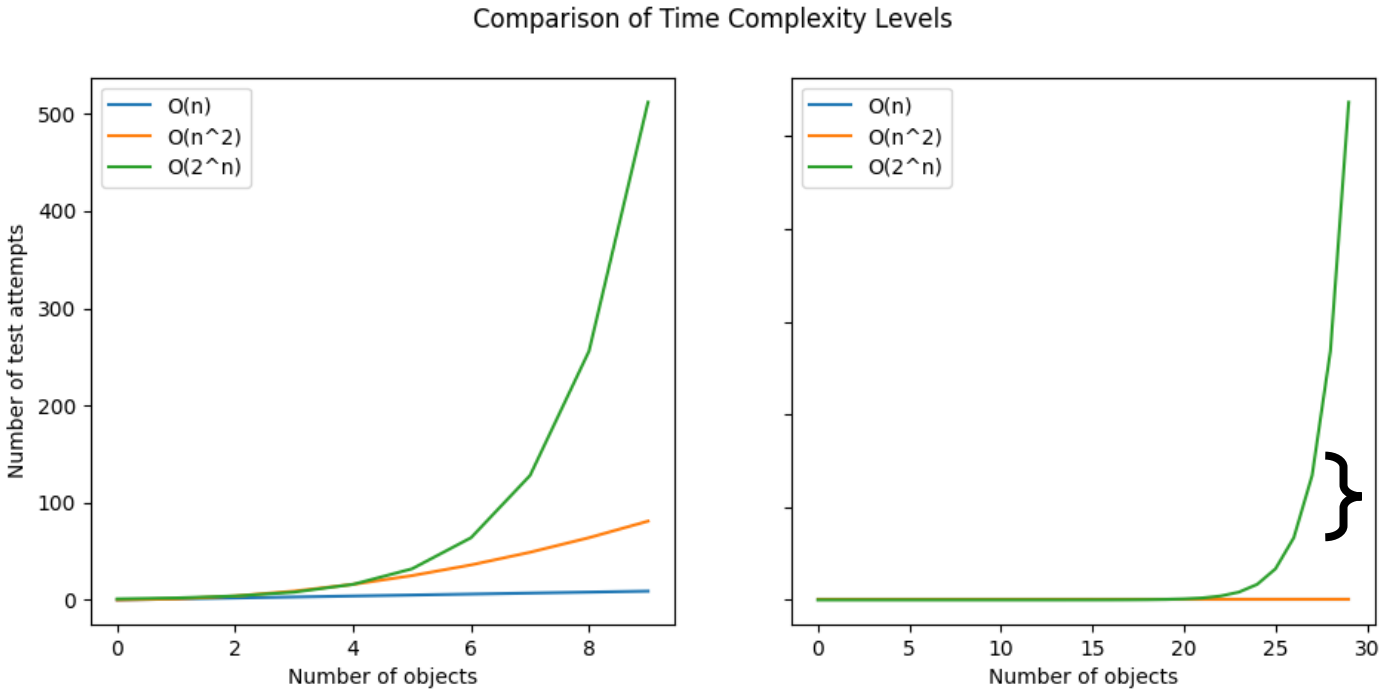
Just one box is a yes/no decision.

Two boxes becomes two yes/no decisions: 4 total choices.

As you can see, the number of cases grow quickly from 2 => 4 => 8 => … => 2^n.

How big is $O(2^n)$?

2^64 is the largest number that the CPU can process natively (without extra steps).
There are about 2^(26~28) stars in our galaxy (assuming 100 billion stars).

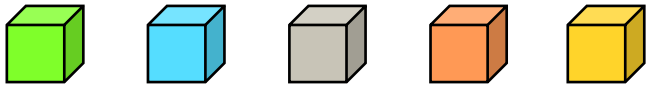


(Estimated)
Number of stars in
Milky Way Galaxy

This is why algorithms other than brute-force are important.

Value/Weight Ratio

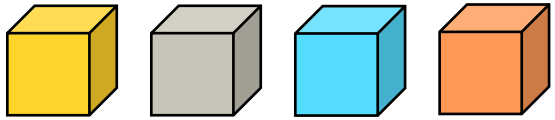
- Dantzig says we should use value-weight ratio, which makes sense because we want to pack valuable but light objects first.



V W	$4 12kg	$2 2kg	$2 1kg	$1 1kg	$10 4kg
V/W	0.33	1.00	2.00	1.00	2.50
Pack Order	5	3	2	4	1

George Dantzig (1914-2005) receiving the National Medal of Science Awards from President Gerald Ford (1913-2006). (The award was given for contribution to linear programming field.)

Final Solution:



8 kg, $15

Can you do better?

Image Source: https://en.wikipedia.org/wiki/George_Dantzig

Analysis of Value/Weight Ratio method

Input

- n objects, each object having properties v (value) and w (weight)
- MW, the maximum weight for the backpack

1. let VW be an array of real numbers
2. let CW ← 0
3. let BAG be an empty array of objects
4. for each object i:
5. $VW_i \leftarrow v_i/w_i$
6. while CW ≤ MW:
7. go through VW:
8. find and add the item with highest VW_i to BAG
9. **answer** with BAG

Why Knapsack is important?

- Knapsack problem is an early, simple *constraint problem*.
- If you can solve this kind of problem, you can learn to solve many practical problems like mathematics, industrial engineering, business modeling, and hardware design.
- It also helps when you want to pack items into a box to send to your home. You want to send high-value merchandise but low weight to cut down on shipping costs.

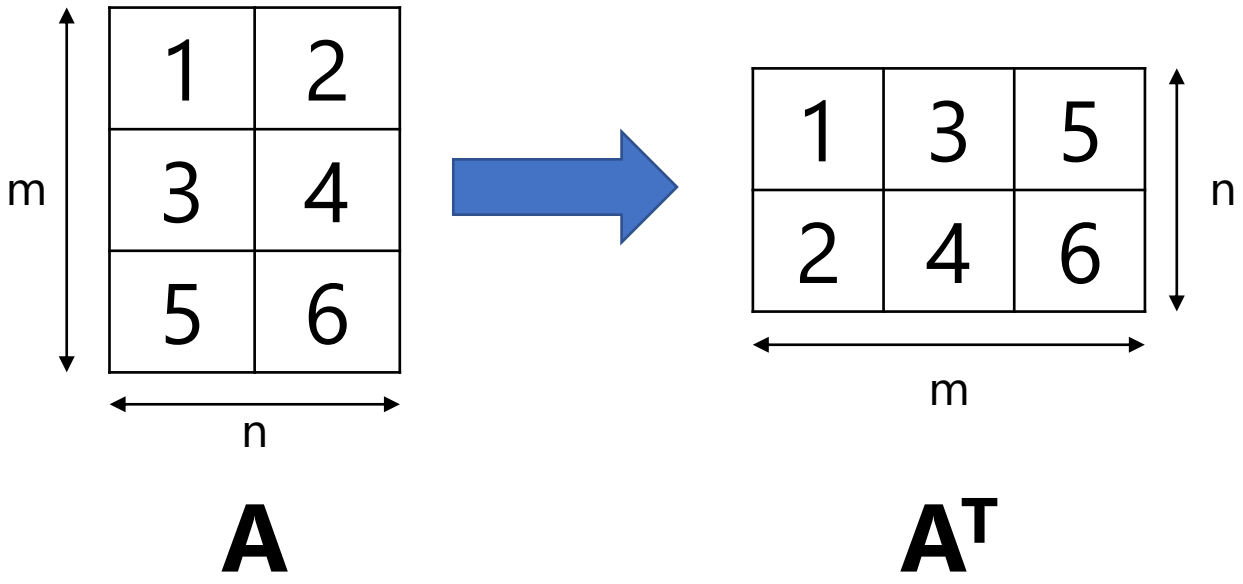
Problem Solving

General Problem-Solving Method

- Determine the input and output information.
- See possible patterns between the input and output.
- Implement an algorithm to solve the problem.
- Write a program (if necessary). You can also use other methods.

Example: Transpose

- Given a matrix A of size (m,n), produce its transpose A^T (should be size n,m).
- $A^T_{ij} = A_{ji}$



Example: Transpose

Input matrix A of size $m \times n$

Output matrix A^T of size $n \times m$

1. create blank matrix AT of size $n \times m$
2. for i ← 0 … n:
3. for j ← 0 … m:
4. $A^T_{ij} = A_{ji}$
5. return A^T (as answer)

Example: Transpose (in Python)

Algorithm (Previous Slide)

Input matrix A of size $m \times n$

Output matrix A^T of size $n \times m$

1. create blank matrix AT of size $n \times m$
2. for i ← 0 … n:
3. for j ← 0 … m:
4. $A^T_{ij} = A_{ji}$
5. return A^T (as answer)

Python Program

```
m = 3
n = 2
A = [[1,2],[3,4],[5,6]]

AT = [[0 for j in range(m)] for i in range(n)]

for i in range(n):
    for j in range(m):
        AT[i][j] = A[j][i]

print(AT)
```

Example: GPA

- Given a list of *n* classes, credits, and grades, determine the GPA for the semester.
- C represents number of credits. G represents grade levels (from 1 to 4 representing C to S).
- GPA is determined by {total of credits and grades product} divided by {total of credits}:

$$\frac{\sum_{i=0}^{n} C_i \times G_i}{\sum_{i=0}^{n} C_i}$$

Example: GPA

Class	Credits	Grade
Mathematics	3	2
Physics	3	3
Programming	3	4
Lab Practice	1	4
Communication	2	3

Input grades table

Output GPA (a real number)

1. let TotalCredits = 0
2. let TotalGradePoint = 0
3. for i ← 0 … n:
4. TotalCredits += C_i
5. TotalGradePoint += C_i * G_i
6. return TotalGradePoint/TotalCredits

Example: GPA
… do you actually need to code?

Class	Credits	Grade
Mathematics	3	2
Physics	3	3
Programming	3	4
Lab Practice	1	4
Communication	2	3

You can spend 3 minutes writing a program to compute your GPA, or just open Excel and type:

=SUMPRODUCT(B2:B6,C2:C6)/SUM(B2:B6)

Today's Summary

- Time Complexity Analysis
- Families of Time Complexity levels
  - O($\sqrt{n}$), O(n), O(n^2), O($n \log n$)
- Algorithms
  - Recursive Algorithms: Factorial and Quicksort
  - Greedy Algorithms: Solving Knapsack Problem
- For problem solving, consider *every* tool and method you have. Just get the job done.
- Next time, we will discuss AI.