Problem Solving

Problem Solving Strategies

Instructor: Jae-Pil Heo (허재필)

Problem Solving Steps

- Step 1: Understand the problem
  - Read the problem carefully.
  - Find the important information.
  - Look for patterns.
  - Identify what the problem wants you to solve.
- Step 2: Decide a strategy to solve the problem
- Step 3: Solve the problem
- Step 4: Validate and analyze your solution

Problem Solving Strategies

- Many problem solving strategies exist:
  - Brute-Force
  - Heuristic
  - Dynamic Programming
  - Reasoning by Analogy (Pattern)
  - Divide & Conquer
  - …

(0/1) Knapsack Problem

- Situation:
  - The next cycle for a given project is 26 weeks
  - We have ten possible products which could be completed in that time, each with an expected number of weeks to complete the project and an expected increase in revenue
  - As a manager, choose those products which can be completed in the required amount of time which maximizes revenue

- This is also called the 0/1 knapsack problem
  - You can place n items in a knapsack where each item has a value in rupees and a weight in kilograms
  - The knapsack can hold a maximum of m kilograms

(0/1) Knapsack Problem

- The products:

Product ID	Completion Time (wks)	Expected Revenue (1000 $)
A	15	210
B	12	220
C	10	180
D	9	120
E	8	160
F	7	170
G	5	90
H	4	40
J	3	60
K	1	10

Knapsack: Brute-Force

- We have $2^N = 2^{10}$ possible choices
  - $\emptyset$
  - $\{A\}, \{B\}, \ldots, \{K\},$
  - $\{A, B\}, \{A, C\}, \ldots, \{J, K\},$
  - $\{A, B, C\}, \{A, B, D\}, \ldots \{H, J, K\},$
  - $\{A, B, C, D\}, \{A, B, C, E\}, \ldots, \{G, H, J, K\},$
  - ...
  - $\{A, B, C, D, E, F, G, H, I, J, K\}$

- For each choice listed above
  - Check that the total required time is less than 26 weeks.
  - If it is a feasible plan, compute the expected revenue and keep the maximum.

Brute-Force

- Brute-Force (Generate & Test)
  - The most obvious way: simply test all the possible cases
  - Always produce the correct answer
  - Often take too much time to run
    - The number of cases are exponentially growing with respect to the problem size.
  - You can use the Brute-Force approach to validate your developed algorithm.
    - Compare the results from your algorithm and ones from the Brute-Force

Knapsack: Heuristic

- Compute the expected revenue per week

Product ID	Completion Time (wks)	Expected Revenue (1000 $)	Revenue Density ($ / wk)
A	15	210	**14 000**
B	12	220	**18 333**
C	10	180	**18 000**
D	9	120	**13 333**
E	8	160	**20 000**
F	7	170	**24 286**
G	5	90	**18 000**
H	4	40	**10 000**
J	3	60	**20 000**
K	1	10	**10 000**

Knapsack: Heuristic

- Sort the products according to the *expected revenue per week*, and select from the top

Product ID	Completion Time (wks)	Expected Revenue (1000 $)	Revenue Density ($/wk)
F	7	170	24 286
E	8	160	20 000
J	3	60	20 000
B	12	220	18 333
C	10	180	18 000
G	5	90	18 000
A	15	210	14 000
D	9	120	13 333
H	4	40	10 000
K	1	10	10 000

Heuristic

- Heuristic/Hill-Climbing/Greedy strategies:
  - Usually take the best choice at the moment
  - Simple and easy to implement

  - Can fail to find the optimal solution
  - Effective on finding a local optimum or a near-optimal solution

Knapsack: Dynamic Programming

- The constraint W (=26 weeks)
- For N products,
  - k^{th} product has a completion time w_k and expected revenue v_k

- Design a recurrence:

$$\text{knapsack}(k,w) = \begin{cases} 0 & k < 0 \\ \text{knapsack}(k-1,w) & w_k > w \\ \max\left(\text{knapsack}(k-1,w), \text{knapsack}(k-1,w-w_k)+v_k\right) & w_k \le w \end{cases}$$

Knapsack: Dynamic Programming

$$\text{knapsack}(k,w)=\begin{cases}0 & k<0\\ \text{knapsack}(k-1,w) & w_k>w\\ \max\left(\text{knapsack}(k-1,w),\text{knapsack}(k-1,w-w_k)+v_k\right) & w_k\le w\end{cases}$$

ID	w_i	v_i
0	15	21
1	12	22
2	10	18
3	9	12
4	8	16
5	7	17
6	5	9
7	4	4
8	3	6
9	1	1

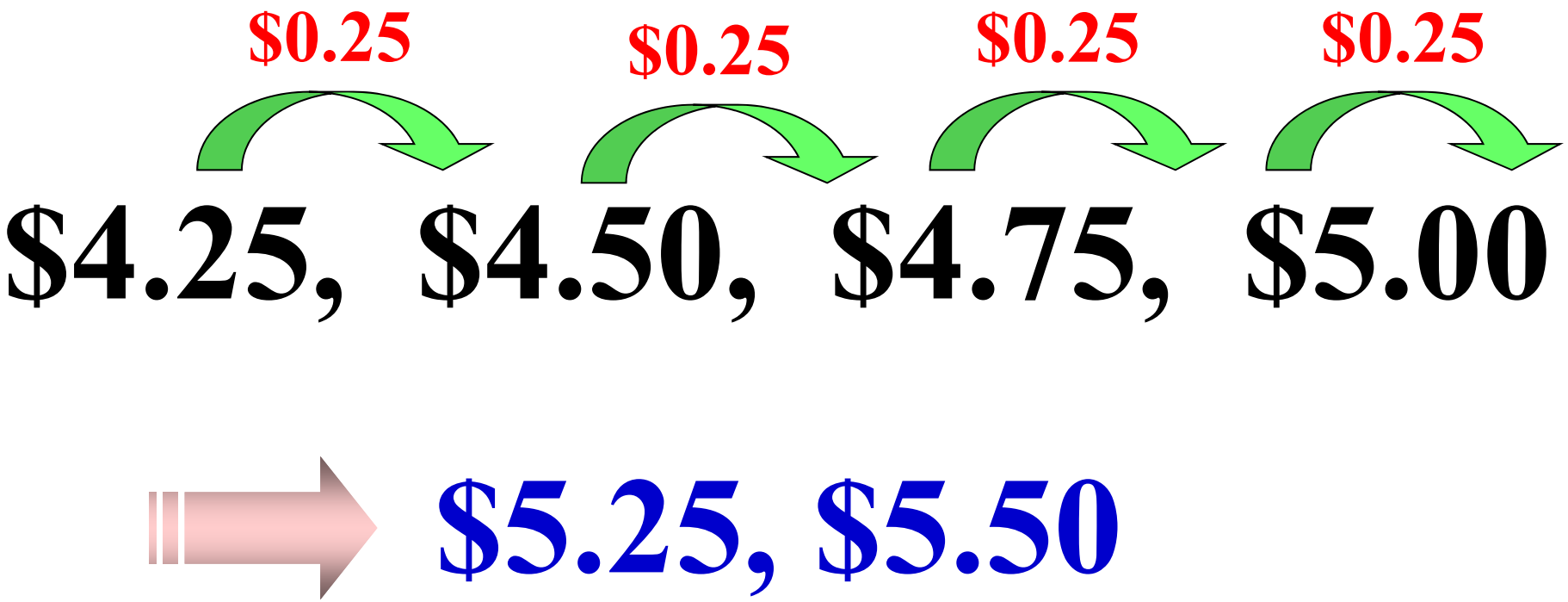
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
#0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0	0	0	0	0	0	0	0	0	0	0
#1	0	0	0	0	0	0	0	0	0	0	0	0	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
#2	0	0	0	0	0	0	0	0	0	0	18	18	22	22	22	22	22	22	22	22	22	22	40	40	40	40	40
#3	0	0	0	0	0	0	0	0	0	12	18	18	22	22	22	22	22	22	22	30	30	34	40	40	40	40	40
#4	0	0	0	0	0	0	0	0	16	16	18	18	22	22	22	22	22	28	34	34	38	38	40	40	40	40	40
#5	0	0	0	0	0	0	0	17	17	17	18	18	22	22	22	33	33	35	35	39	39	39	40	40	45	51	51
#6	0	0	0	0	0	9	9	17	17	17	18	18	26	26	26	33	33	35	35	39	42	42	44	44	48	51	51
#7	0	0	0	0	4	9	9	17	17	17	18	21	26	26	26	33	33	35	35	39	42	42	44	44	48	51	51
#8	0	0	0	6	6	9	9	17	17	17	23	23	26	26	27	33	33	35	39	39	42	42	45	48	48	51	51
#9	0	1	1	6	7	9	10	17	18	18	23	24	26	27	27	33	34	35	39	40	42	43	45	48	49	51	52

Dynamic Programming

- Useful when recursive algorithms have overlapping sub-problems
- Storing calculated values (using a table) allows significant reductions in time
  - Memoization
- Hard to be applied to some cases such as non-integral data

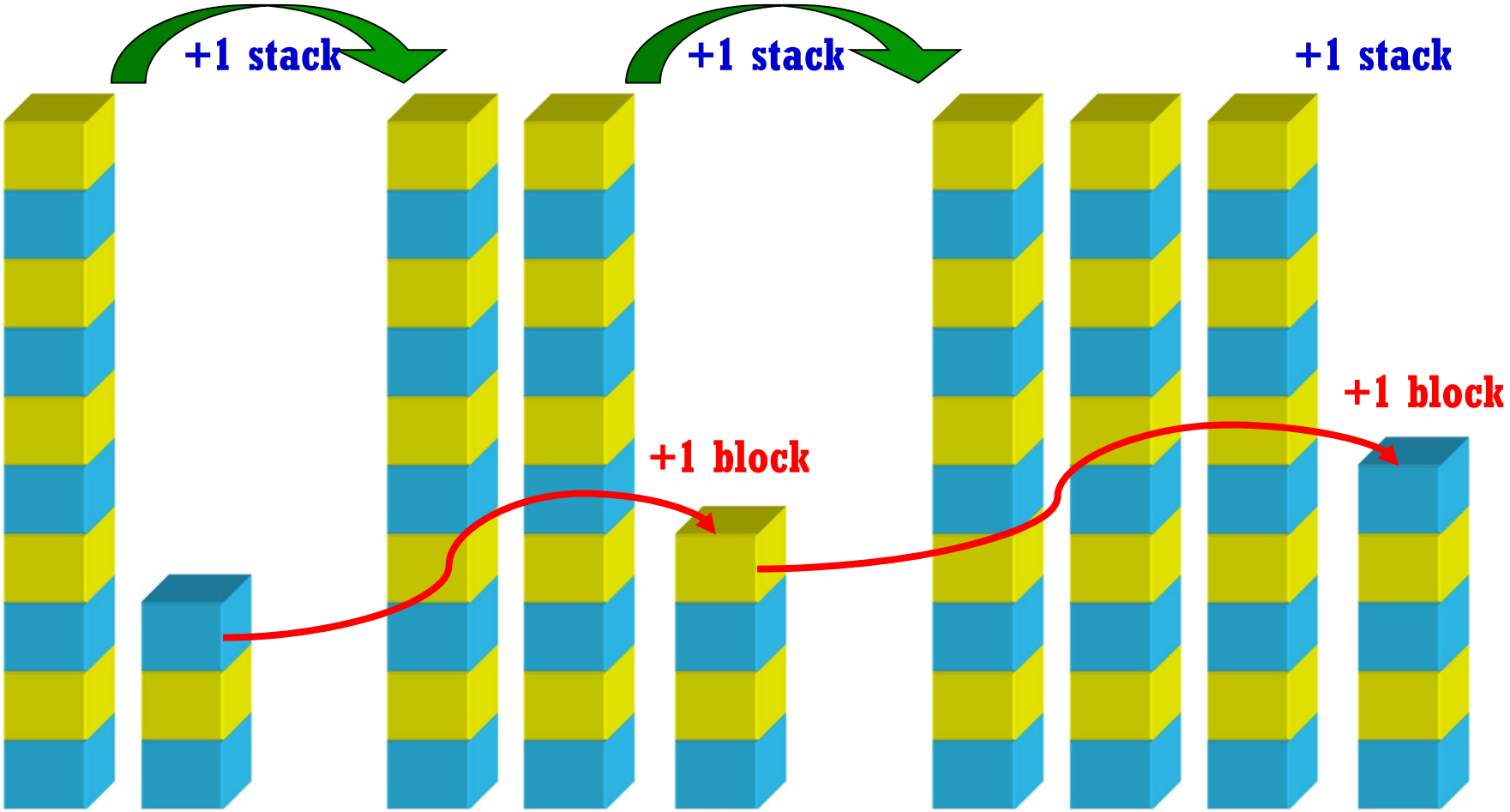
Reasoning By Analogy (Pattern)

- What are the next two numbers?



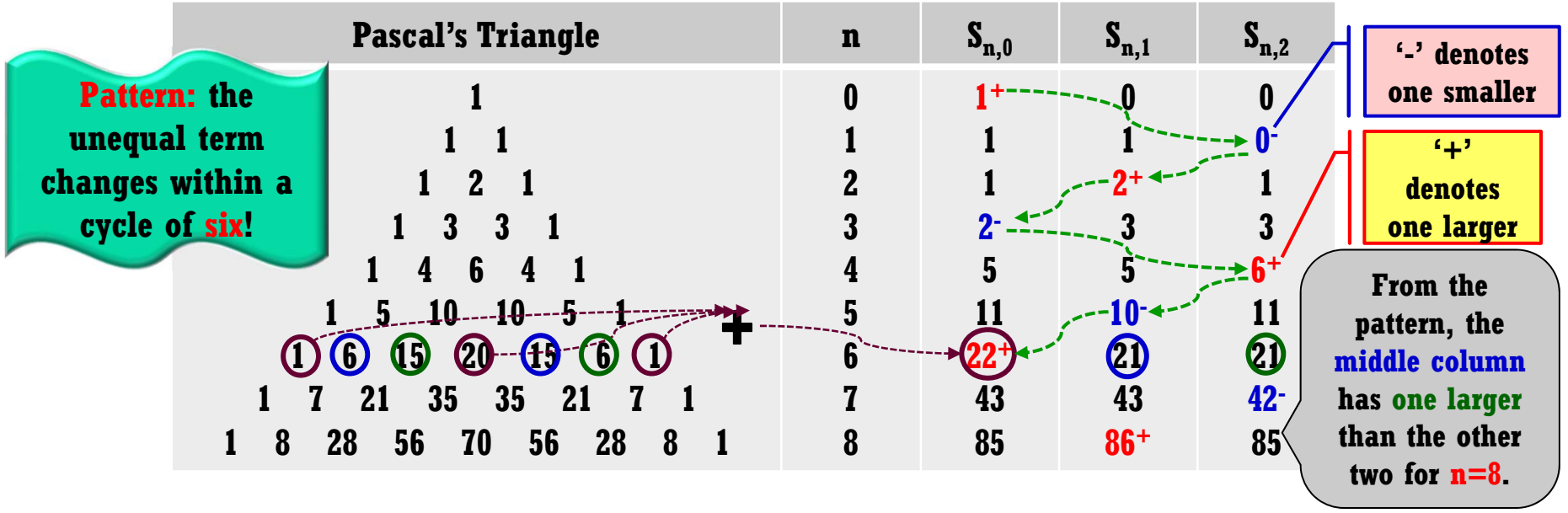
Reasoning By Analogy (Pattern)

- What are the next set of blocks look like?



Reasoning By Analogy (Pattern)

- Let $S_{n,0}, S_{n,1}$, and $S_{n,2}$ denote the sum of every third element in the n^{th} row of Pascal's Triangle, beginning on the left with the first, the second, and the third element, respectively.
- What is the value of $S_{100,1}$?



Reasoning By Analogy (Pattern)

- We have $S_{n,0} + S_{n,1} + S_{n,2} = 2^n$
- Since $100 = 6 \times 16 + 4$, the unequal term appears in the third column $S_{100,2}$
- Since $S_{100,0} + S_{100,1} + S_{100,2} = 2^{100}$, $S_{100,1} = \frac{2^{100}-1}{3}$

Pattern: the unequal term changes within a cycle of six!

Pascal's Triangle

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
.....
1 100 100
1

n	$S_{n,0}$	$S_{n,1}$	$S_{n,2}$	$\sum S_{n,i}$
0	1^+	0	0	1
1	1	1	0^-	2
2	1	2^+	1	4
3	2^-	3	3	8
4	5	5	6^+	16
5	11	10^-	11	32
6	22^+	21	21	64
7	43	43	42^-	...
8	85	86^+	85	...
.....	.....	.....	.....	2^n
100	$S_{100,0}$	$S_{100,1}$	$S_{100,2}$	

Same pattern

The third column has one larger than the other two for n=100.

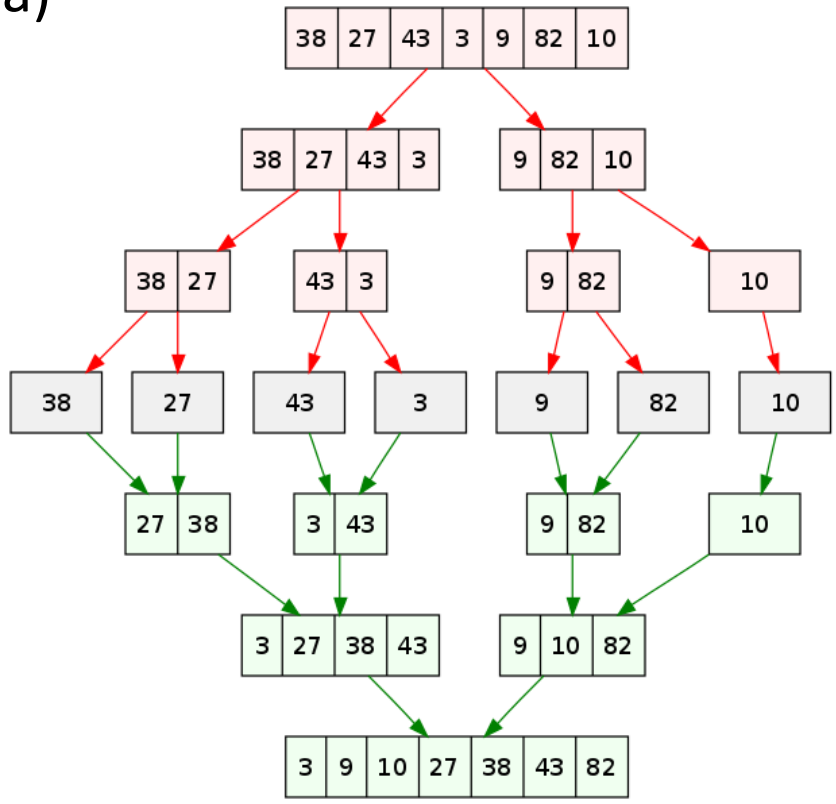
Divide and Conquer

- *Divide* problem into several smaller problems
  - Normally the subproblems are similar to the original
- *Conquer* the subproblems by solving them recursively
  - Base case: solve small enough problems by brute force
- *Combine* the solutions to get a solution to the subproblems
  - And finally a solution to the original problem

- Efficient for many real-world applications
  - Sorting, searching
- Suitable for the parallelization
  - Multi-threading and cache efficiency

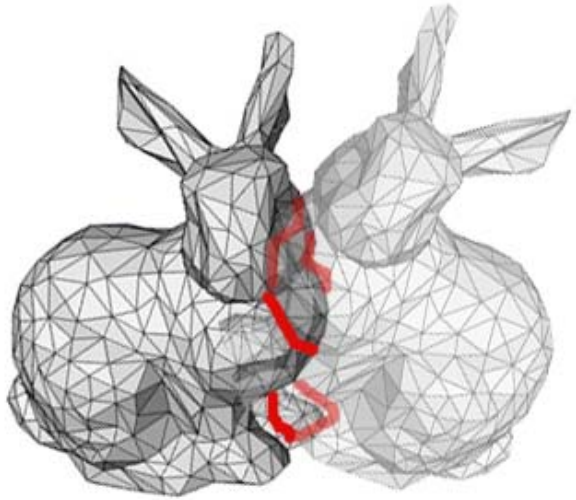
Divide and Conquer: Merge Sort

- Merge sort
  - Divides the unsorted list into n sublists, each containing 1 element, and repeatedly merges sublists to produce new sorted sublists until there is only 1 sublist remaining. (Wikipedia)



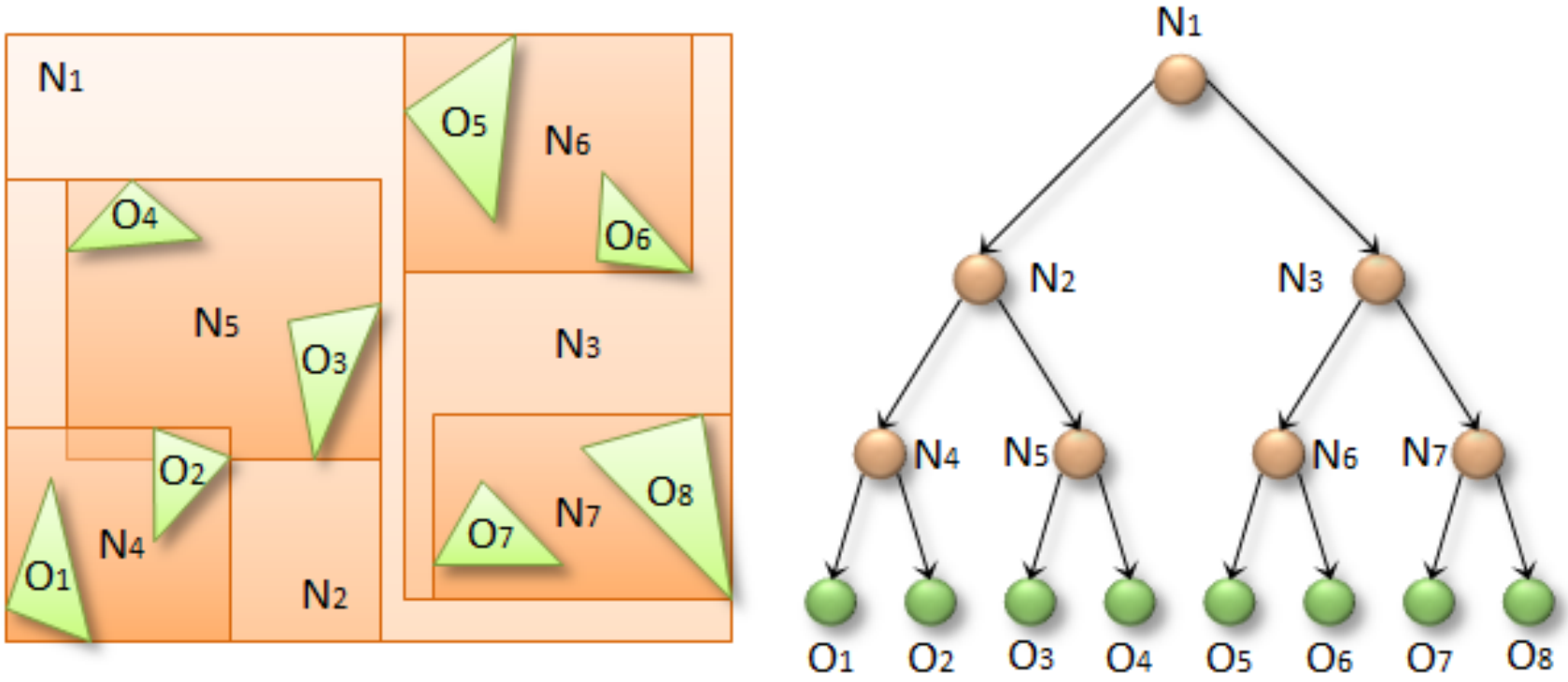
Collision Detection (In CG)

- Identify the intersection of 3D models
- Used in various applications including games, physically-based simulation, and robotics.

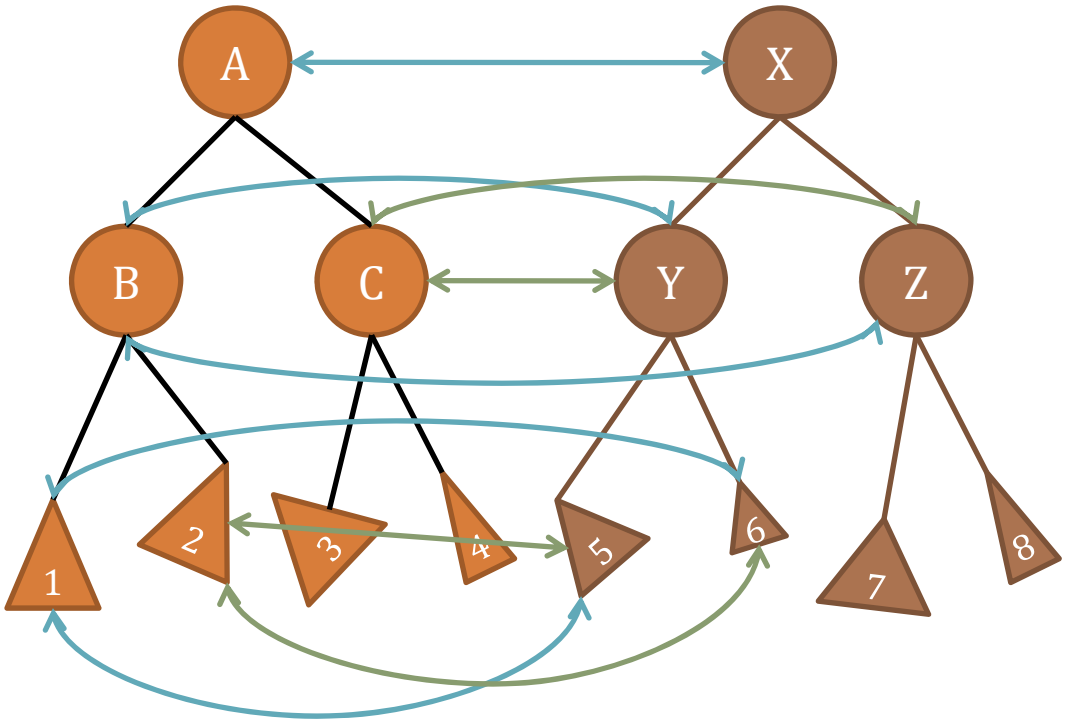


- A naïve approach
  - Perform intersection tests for every pair of triangles.

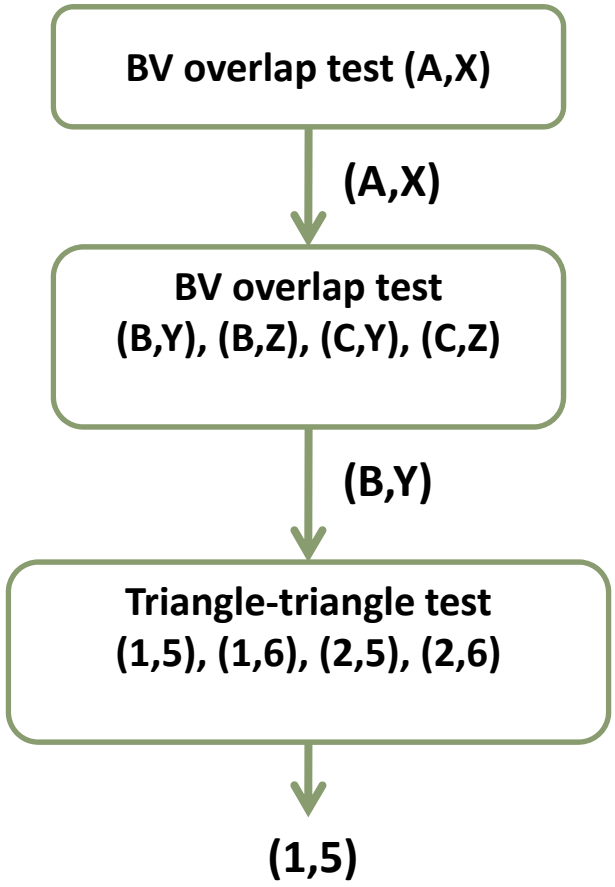
Bounding Volume Hierarchy



Divide and Conquer: Collision Detection



Triangle 1 and 5 intersect!



Divide and Conquer vs. Dynamic Programming

- Both approaches divide the given problem into subproblems and solve the subproblems.

- How to choose one of them?
  - Divide and Conquer should be used when same subproblems are not evaluated many times.
  - Otherwise, Dynamic Programming with Memoization should be used.

- Example:
  - In Merge Sort, we never evaluate the same subproblem again.
  - In Fibonacci number, Dynamic Programming should be preferred.

Any Question?