PRE-FINAL REVIEW

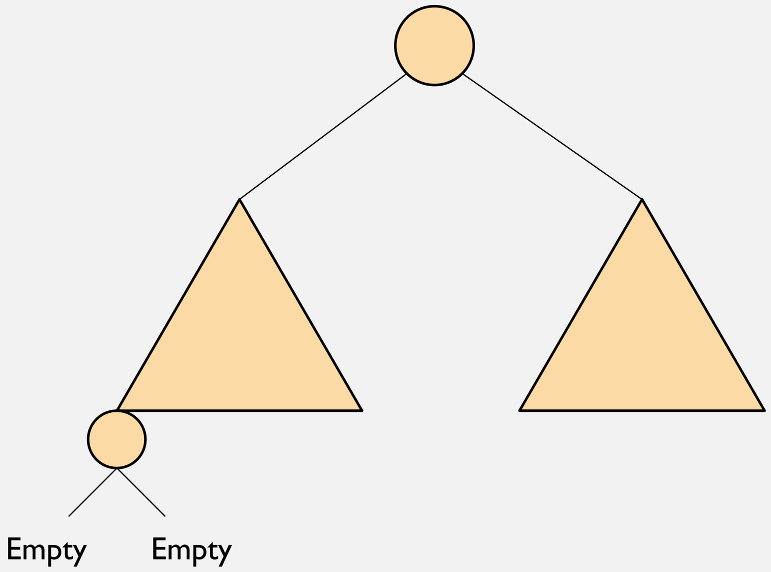
2024-04-18

INCLUDED TOPICS

- 8 problems
- 2 pages of an A4 cheat sheet allowed
- Calculator allowed
- Contents
  - Graph algorithm: recursive algorithm for binary tree // max-flow min-cut
  - Greedy algorithm: proof of the loop invariant// find counter example// fix the code
  - Dynamic programming algorithms: fill the tables
  - Parallel algorithms: analyze time complexity

GRAPH ALGORITHMS

NUMBER OF NODES

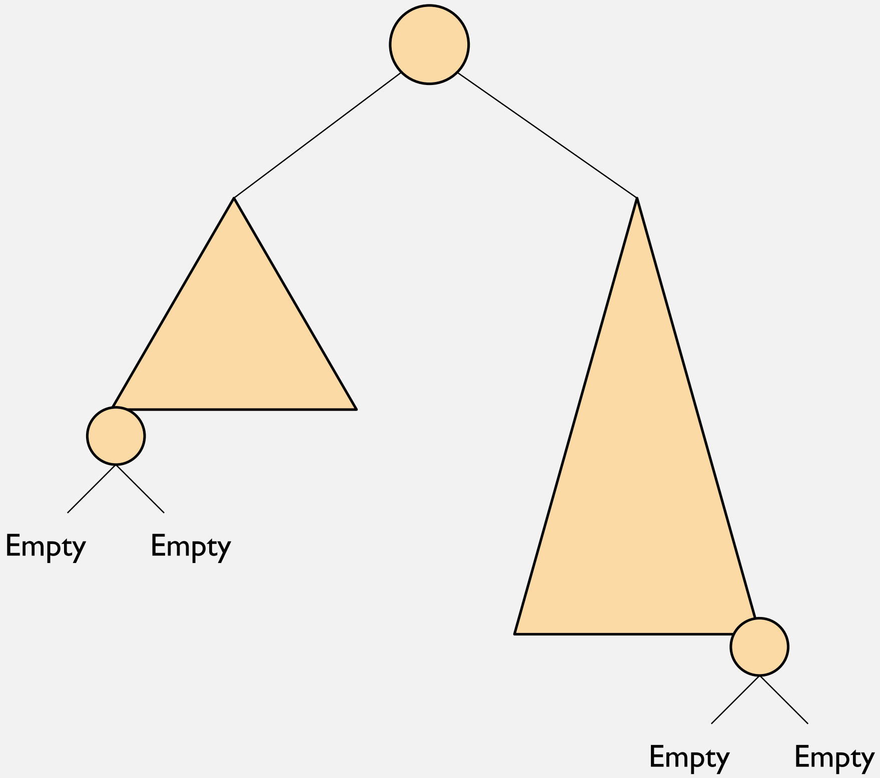


- Don’t want to count empty nodes
- Ask left friend and right friend to find their number of nodes.
- Extra work:

$num(tree)\ =$

- Simplest case:
  - When the input is an empty tree, $num(empty) =$

HEIGHT OF THE TREE

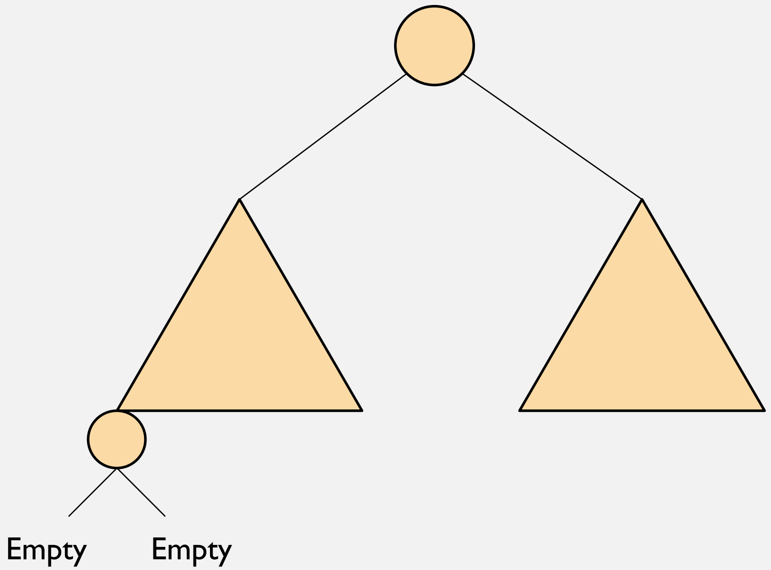


- Note: tree with 1 node has height = 0.
- Ask left friend and right friend:
- Extra work:

$height(tree) \; =$

- Simplest case:
  - When the input is an empty tree $height(empty) =$

NUMBER OF LEAF NODES



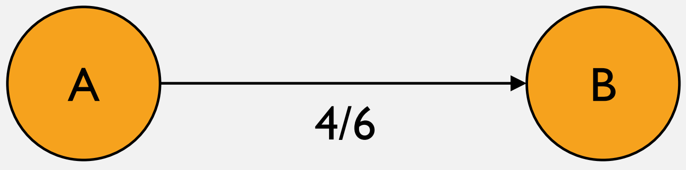
- Leaf node must be non-empty
- Ask left friend and right friend:
- Extra work:

$leaves(tree) \ =$

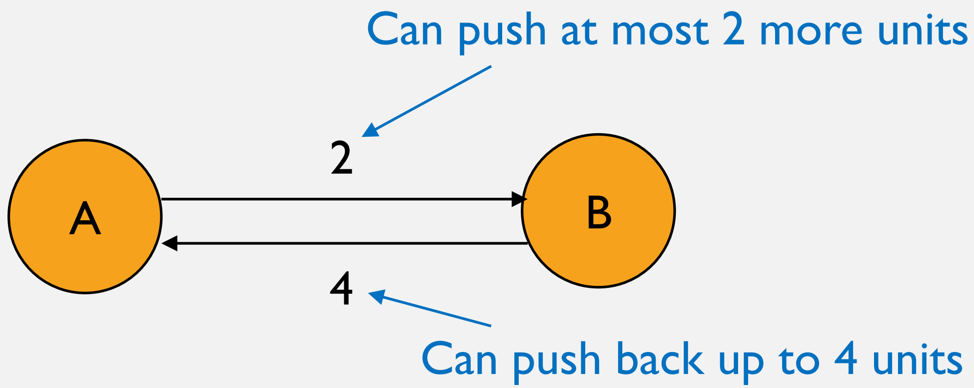
- Simplest case:

RESIDUAL GRAPH

- The new available path can be
  - A totally new path with edge capacities that have never been used before.
  - A path that reduces flow in some edges of the existing flow.



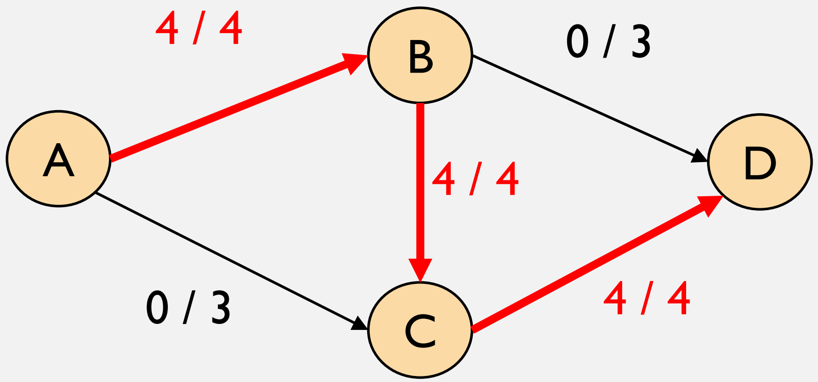
Current flow



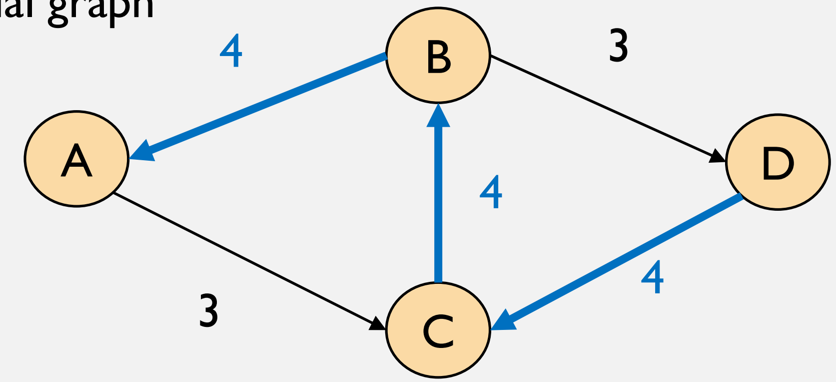
Residual graph

RESIDUAL GRAPH

Current flow



Residual graph

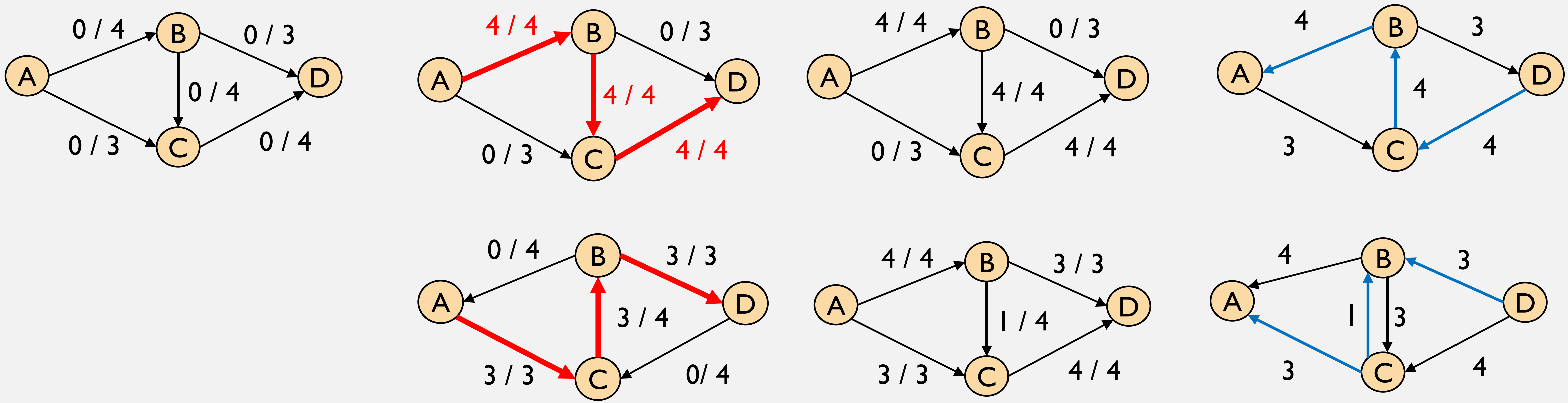


- Residual graph provides the availability of the remaining capacity, or the capacity that has been used and can be reduced (equivalent to available flow in the opposite direction).

- Do you see more available path from the residual graph?

STEP BY STEP

Original



New path found

Total flow being used

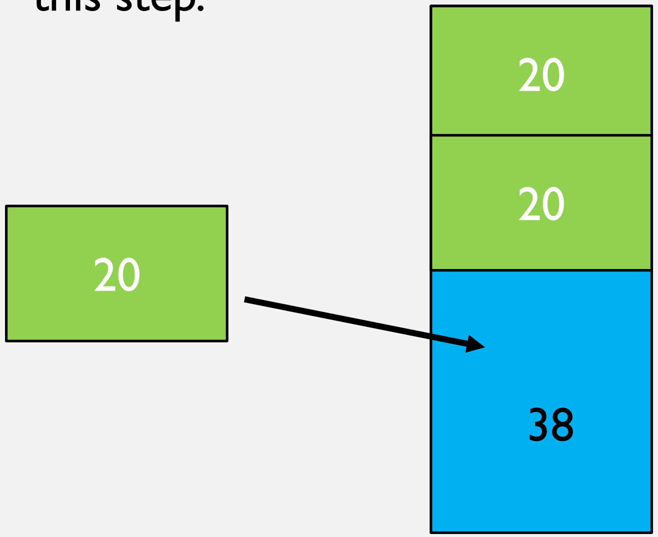
Residual graph

A and D are disconnected

GREEDY ALGORITHMS

(1) Algorithm:

Greedy algorithm chooses 20 baht token in this step.



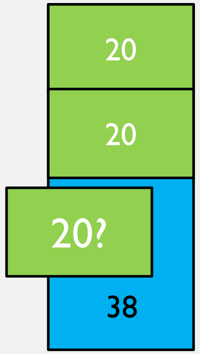
(2) Fairy godmother:

I have an optimal solution that extends your choice [20, 20] in the previous step, but I won't reveal the entire solution to you.

(3) Prover:

I will make sure that the current algorithm's choice is not worse than any optimal solution of the fairy godmother, no matter what she is hiding.

ADJUST S_{t-1} SOMEHOW, MAKE IT COMPATIBLE WITH A_t.

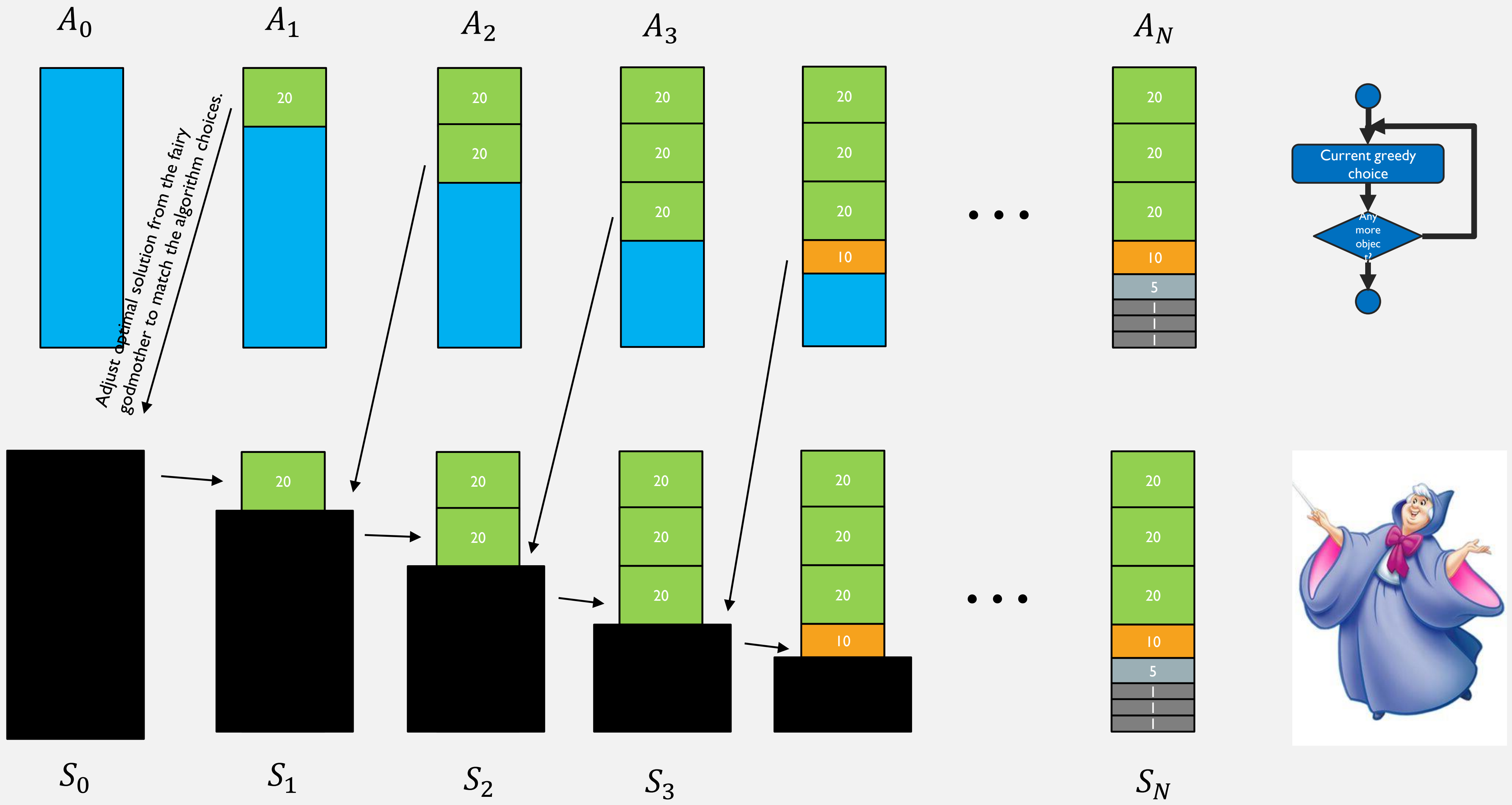


Algorithm choose another 20 baht token

Fairy godmother optimal solution S_{t-1}

If the remaining value is greater than 20, we have to show that the algorithm choice (choosing a 20 baht token) is the best choice.

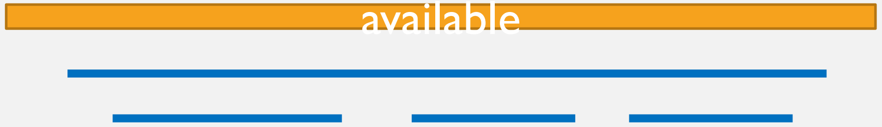
Then adjust S_{t-1} to use a 20 baht token as well. After the adjustment, S_{t-1} with one more 20 bath token becomes S_t.



No matter what the optimal solution has under the cover, the greedy algorithm choices is not worse than that.

PRIORITY FUNCTION THAT WORKS

- Consider the ending time.
- Choose the events that does not conflict with the existing event, and finish earlier than others.



WHY DOES THAT WORK?

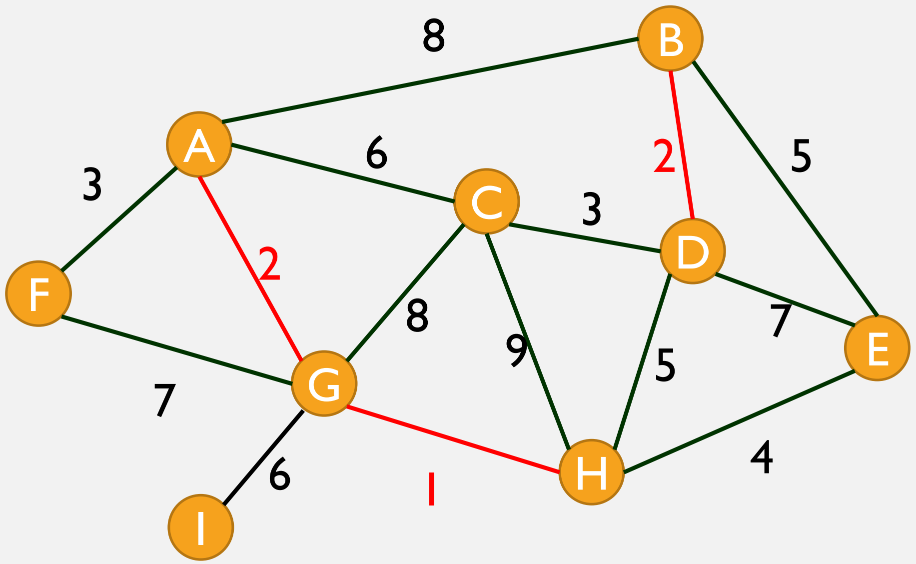
- The event chosen by the algorithm ends first among all the rest.
- When compared to the first event in her unrevealed event sequences, the algorithm choice ends before or at the same time as her.
- The replacement does not affect the ending time of the current event (because it finishes at the same time or faster). All other events in her solution does not need to change.

- So the modified solution is still optimal (take one out, put one in)
- The replacement make it consistent to A_t
- The modified solution is valid since the current event makes no conflict with any other events in the fairy godmother solution.

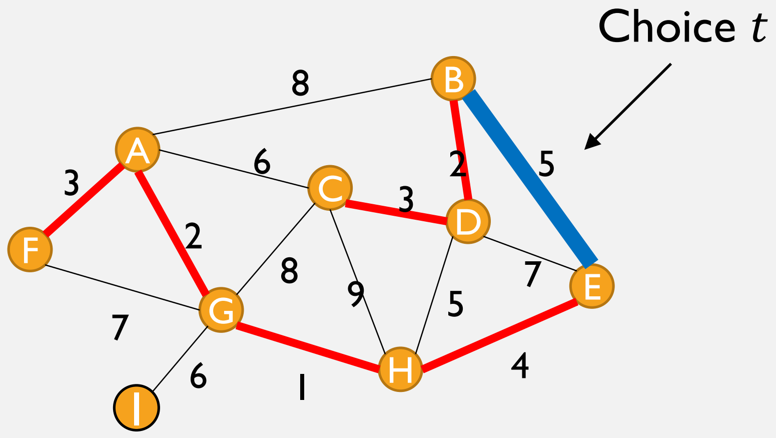
I agree with that

KRUSKAL'S ALGORITHM

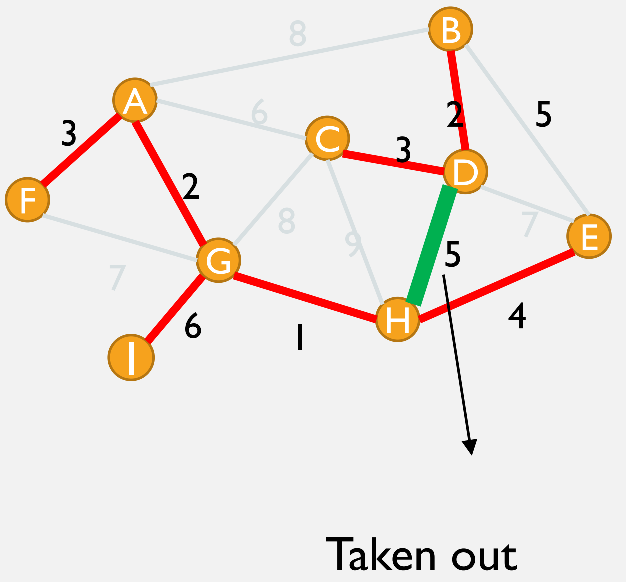
- The newly added edge is the minimum weight edge that does not create a cycle.



A_t



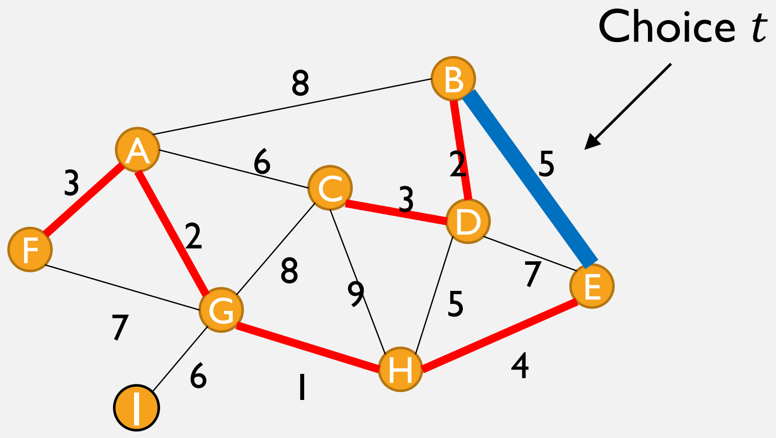
S_{t-1}



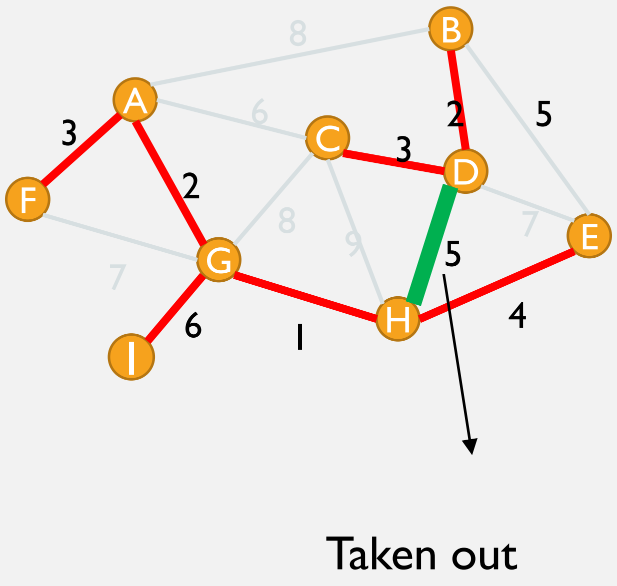
- A_t and S_{t-1} might be different.
- Prover has to offer the new choice in loop t so that the fairy godmother agree to use.

- Since S_{t-1} is an optimal solution, it is a spanning tree. Adding one edge into it will create a cycle. So she has to take one edge out in order to preserve the validity (not to have any cycle).
- Case 1: If the current choice t are the same as in S_{t-1}, there is nothing to modify.
- Case 2: The edge to be taken out is not the same as the current choice.
  - Then there will be a cycle.
  - Take out the edge with highest weight in the cycle (but not the one we just added) out.

A_t



S_{t-1}

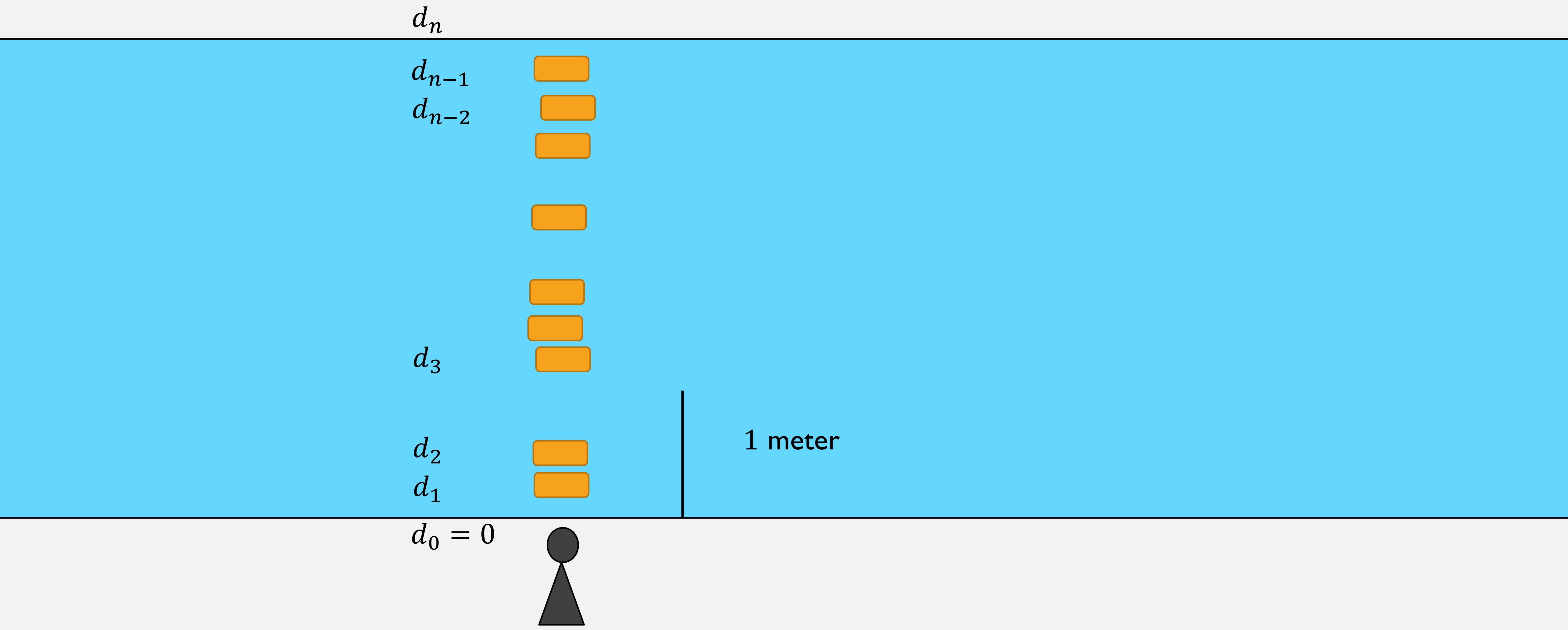


- Take out the edge with highest weight in the cycle (but not the one we added) out.
- This particular edge has the weight not less than the one we added, otherwise, the algorithm should have chosen this edge before the current choice.
  - (Since the algorithm process the edges according to the order of the weights. If the one to take out has the smaller weight, the algorithm must have made a decision on that edge before this step)

- Therefore, the weight of the added edge is less than or equal to the edge to be removed.
- We can maintain the optimality of the spanning tree.

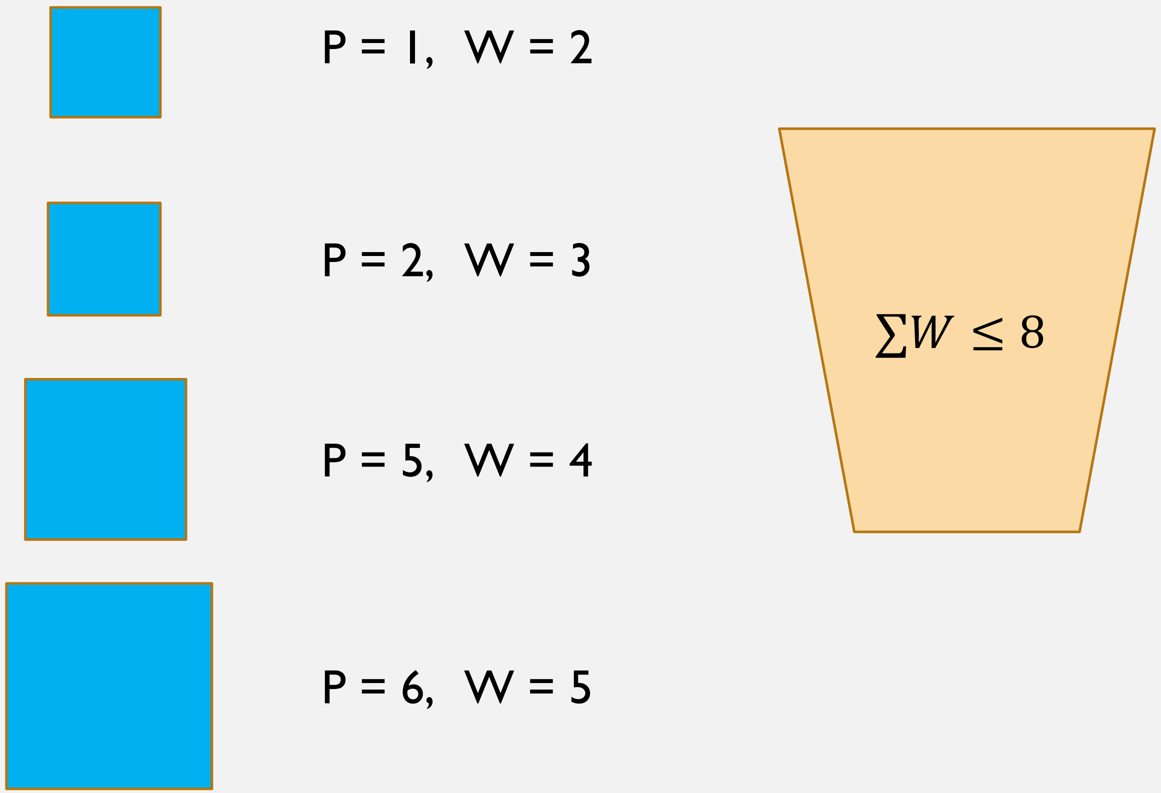
IS IT OK?

- **Valid**
  - The added edge creates a cycle, then take one edge in the cycle out. The entire subgraph is still connect and is without loop. So it is a spanning tree.
- **Optimal**
  - Weight of the added edge is less than or equal to the edge to be removed.
- **Consistent** with A_t
  - We put the algorithm choice into S_t.


d_n
d_{n-1}
d_{n-2}
d_3
d_2
d_1
$d_0 = 0$
1 meter

DYNAMIC PROGRAMMING ALGORITHMS

0/1 KNAPSACK PROBLEM



- We have a bag with capacity of 8 kg.
- There are 4 objects with different weights and profits.
- 0-1 means that you can either choose or not choose the object, not cut in a half.
- We want to carry objects so that the total weight does not exceed the bag capacity, and the total profit is maximized.

NEXT ROW (FIRST OBJECT)

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2					2				
3									
4									

+2

	-	C	H	A	R	T
-	0	0	0	0	0	0
C	0	1	1	1	1	1
A	0	1	1			
S	0					
T	0					

- Not skip at all $(A = A)$
    - Refer to CH and C
    - Add one matching $(A = A)$
- Skip for the first sequence
    - Refer to CA and CH
- Skip for the second sequence
    - Refer to CHA and C

MAIN EQUATION

$$opt_{i,j} = \max(opt_{i-1,j}, opt_{i,j-1}, opt_{i-1,j-1} + 1\ )$$

Exists only when $s_1[i] = s_2[j]$

	-	C	H	A	R	T
-	0	0	0	0	0	0
C	0	1	1	1	1	1
A	0	1	1	2		
S	0					
T	0					

$$A_{2\times 2} \times B_{2\times 3} \times C_{3\times 1} \times D_{1\times 4}$$

	A	B	C	D
A	0	12	10	
B		0	6	
C			0	12
D				0

$$(A \times B \times C)$$

$(A \times B) \times C$

Total cost $= 18$

$A \times (B \times C)$

Total cost = 10

	A	B	C	D
A	0	0	0	
B		1	1	
C			2	2
D				3

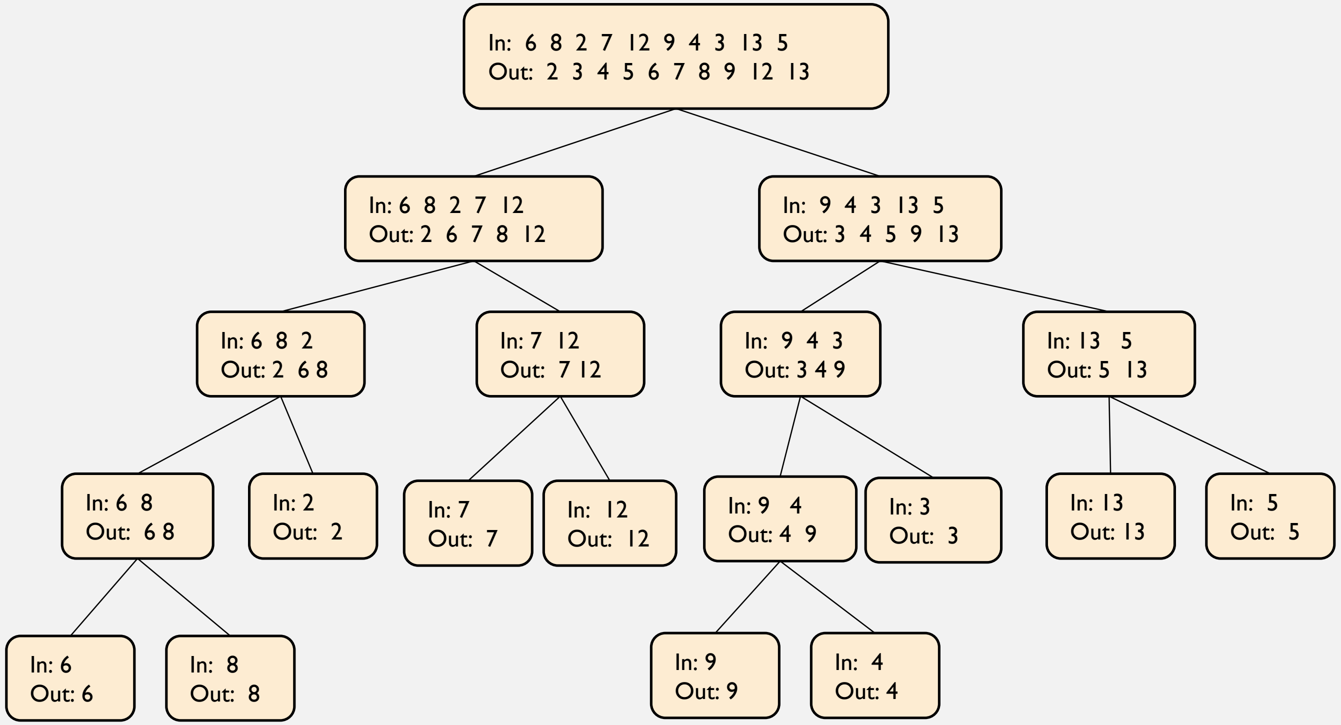
The end of the first partition is the matrix index 0

PARALLEL ALGORITHMS

PARALLEL BUBBLE SORT

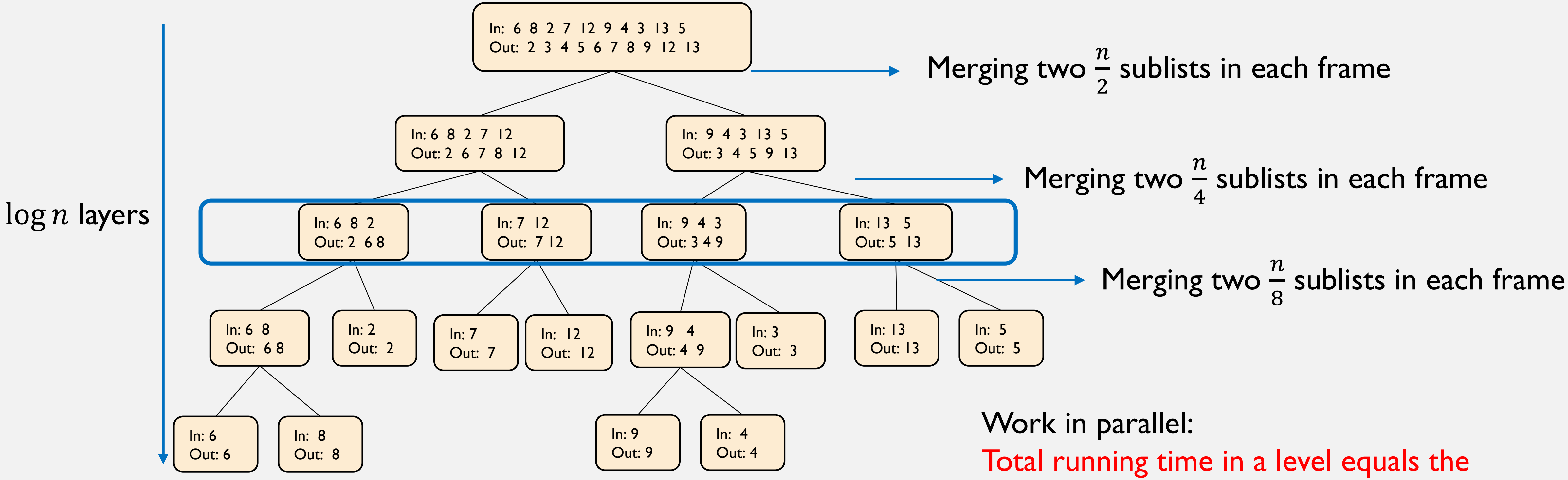
- Also called <u>odd-even sort</u> or <u>brick sort</u>
- If the second pair must wait for the result of the first pair, we can do compare and exchange at the third pair
- While working with the third pair, we can work with the 5^{th} pair since there is no overlap between them
- Parallel bubble sort assign "compare and exchange" to all of the odd pairs at the same time
- Then the next iteration will be done on the even pairs
- So we can work with $\frac{n}{2}$ pairs at once in just 1 unit of time.

PARALLEL MERGE SORT



- Recall: Merge sort is a recursive sorting
- It seems like assigning friends to help, but friends are just a copy of ourselves.
- Where should we ask multi-processor to do in parallel?

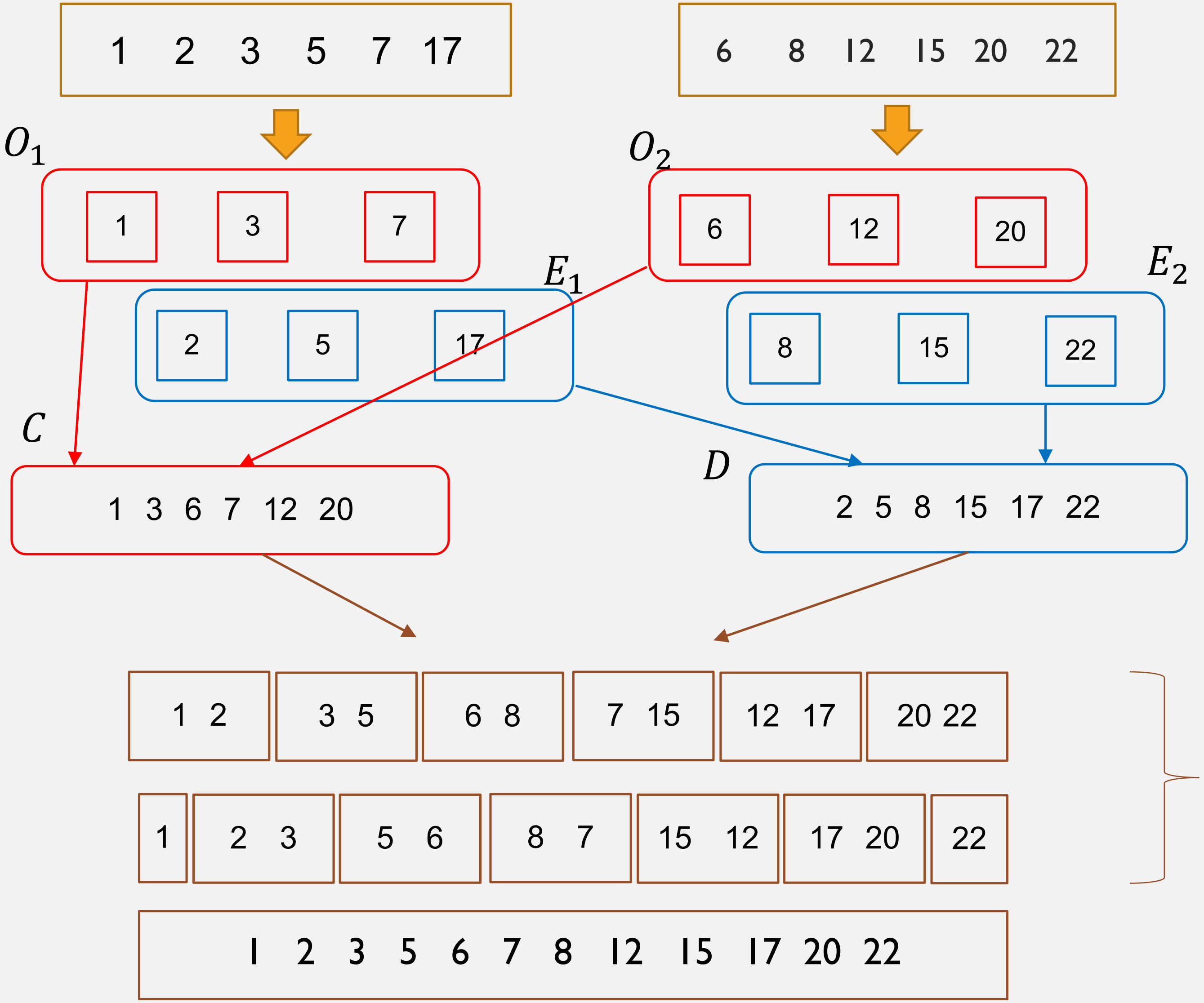
TIME COMPLEXITY



Work in parallel:

Total running time in a level equals the running time of just one processor

ODD-EVEN MERGE



- Generate O_1, O_2 and E_1, E_2 sublists, containing ODD and EVEN objects from sublist1 and sublist2
- Merge O_1 and O_2 into C, and merge E_1 and E_2 into D by using the recursive parallel merge (this algorithm itself).
- Claim that C and D now are in the good shape for assigning to parallel processor to merge.
- Perform odd-even sort for 2 iterations.
  Then they are merged successfully.

ODD-EVEN MERGE SORT

- We then use odd-even merge to replace the traditional merging in the merge sort.
- There are $\log n$ levels in the merge sort.
- Each level must perform odd-even merging in parallel, which takes $O(\log n)$ time.
- Overall odd-even merge sort takes $\boldsymbol{O(\log^2 n)}$