Greedy algorithm

Fall 2024

- Method for solving optimization problems.
    - Problems which requires a **minimum** result or **maximum** result.
- For a given problem P, there may be multiple solutions: S1, S2, S3…
- Suppose you want to go from A to B.
    - Multiple ways to solve this.
    - Walk, run, bus, fly, etc.
- There are multiple solutions, but every solution has some time.
    - Some are faster, some are comfortable, etc.
- Sometimes (often) there is a **constraint**
    - I need to cover the journey in **x** hours.
        - Some solutions are not available any more.

- Sometimes there is a **constraint**
    - Some solutions satisfy this constraint.
    - Some solutions do not.
- The solutions which **satisfy** the constraint is called: **Feasible solutions**
    - Solutions satisfying the constraint.
- What if we want to complete this journey in **minimum cost**?
    - Now, it becomes a **minimization problem** → Optimization problem
    - Out of **feasible solutions** one of them can be minimum.
        - Train vs plane
        - If train is the minimum solution → we call it **optimum solution**
- Optimum solution:
    - Solution which is already **feasible** and also **satisfies** the objective given (minimum cost).
- There is only **one** optimal solution.
    - One minimum cost.

Greedy method is used to solve optimization problems

- If a problem requires either **minimum** or **maximum** results,
- Those kind of problems are called
  - Optimization problem
- Summary
  - There are multiple solutions to a problem.
  - Sometimes there is a constraint. Now, the solution set becomes smaller.
  - The solutions which satisfies that constraint are called **feasible solutions**
  - In addition, we may have an **objective**
    - That objective is usually looking for something **min** or **max**
    - Minimum cost, maximum profit, minimum time, etc.
  - Out of those solutions, if a solution satisfies this **objective** → That is called the optimal solution.
  - The problem with a constraint and an objective is called an **optimization problem**

- There are several strategies to solve optimization problems.
  - Greedy method
  - Dynamic programming
  - Branch and bound
- Some problems are suitable for some methods.

Greedy method

- In greedy method, a problem is solved in **stages**
- In each stage, we consider one input of a problem.
    - If that input is *feasible* we include it in the solution
    - After including all feasible inputs, we will get an optimal solution.
- We use it in daily life!
-

Examples

- Coin change
  - Give customer change, using **minimum number of coins**
  - 1, 5, 10, 20, 50
    - You will start by giving the largest first. (strategy)
  - Amount: 87
    - 50, 20, 10, 5, 1
- Buying items
  - Fixed budget, but want to buy **as many items as possible**
    - So, since the objective is to buy *maximum number of items* you start with cheapest goods. (strategy)
- Packing
  - You have a bag (limited space) but must pack items.
    - Go for the *importance.* (strategy)
- Charging
  - You have multiple devices with varying battery percentages and limited time.
    - Go for **lowest battery** first or the one **you need** (strategy)

Constraint & Objective

Task	Constraint	Objective
Coin change	value = required amount	Minimum number of coins&bills
Buying items	Total cost <= budget	Maximize the # of items
Packing	weight <= desired weight	maximize the importance or value of items
Charging devices	limited time or ports	Max. the number of ready devices

Knapsack problem

Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1

- Knapsack problem is a bag problem.
  - There is a bag and we want to fill it with items.
  - There is a **limit** → 15 kg. (constraint)
  - We want to **maximize** the profit. (objective)
- Imagine that we are filling the bag and going to another city, to sell those items.
  - Profits are those.
  - Weights are weights.
- Container loading problem.
  - An optimization problem.
- There are multiple solutions.
  - I can just put items 1 and 2.
  - It is a solution, and a *feasible* solution → but not the optimal solution.
  - Our aim is to find the optimal solution.

Knapsack problem

Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1

- What is going to be our strategy?
    - Someone can say let’s start with highest profits!
    - Some other person can say; let’s start with the **lightest** objects. So we can get more stuff it.
- Both are valid.
    - However, we need a strategy.
    - What about **profit per kg** (imagine that weight is in kg’s. not important)
- So we need to calculate **profit per kg**

Knapsack problem

Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1
P/W:	5	1,67	3	1	6	4,5	3

- Now, we have a **P/W**.
  - We can use it to put items in the knapsack.
- This problem is not a 0/1 knapsack problem.
  - In that problem, we cannot use fractions.
  - But here we can use fractions. Maybe I am going to use half of object 3.
    - I can choose to buy half kg of something right?
- So, we start by selecting the object with **highest** P/W.
- Which element has the best PW ?
  - Object 5.
  - I add it to the bag.
  - I had a constraint of 15 kg. Object 5 is 1 kg → 15-1 = 14.
  - I now have 14 kg remaining in the bag.

Knapsack problem

Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1
P/W:	5	1,67	3	1	6	4,5	3

- Bag: [Object 5]
- I look at the next highest PW.
    - It is object 1.
    - I add it to the bag.
    - Bag weight was 14 kg. 14 - 2 = 12 → new weight
- Bag: [O5, O1]
- Next highest?
    - Object 6.
    - Add it to bag.
    - Weight was 12. 12 - 4 = 8 → new weight
- Bag: [O5, O1, O6]
- Next highest?
    - O3 and O7. (Choose 1) → O3
    - 8 - 5 = 3 → new weight
- Bag: [O5, O1, O6, O3]
- Next highest?
    - O7.
    - 3 - 1 = 2 → new weight
- Bag: [O5, O1, O6, O3, O7]
- Next? → O2
    - 2 - 3 ?? → can't do it.
    - But we can do **fractions**
    - We use 2 kg of O2.
- Bag: [O5, O1, O6, O3, O7, ⅔ O2]

Knapsack problem (Profit)

Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1
P/W:	5	1,67	3	1	6	4,5	3

- Bag: [O5, O1, O6, O3, O7, ⅔ O2]
    - I used O5 → 6 profits
    - I used O1 → 10 profits
    - I used O6 → 18 profits
    - I used O3 → 15 profits
    - I used O7 → 3 profits
    - I used ⅔ O2 → 5*⅔ profits
- **Total:**
    - 6+10+18+15+3+10/3
    - 55,3 total profit

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- Imagine that you are working an in the morning there are multiple jobs waiting for you.
  - We can change the question so that it is more meaningful.
  - Let’s say you are a typist. These are the jobs. You start the day at 0900 am.
  - Job 1 tells you that they will come at 11.00 ➝ They will pay you 20.
  - Job 2 tells you that they will come at 11.00 ➝ They will pay you 15.
  - Job 3 will come at 1000 ➝ Pay you 10
  - Job 4 will come at 1200 ➝ Pay you 5.
  - Job 5 will come at 1200 ➝ Pay you 1.
- For simplicity, let’s imagine all jobs will take an hour ➝ 1 unit time.
- Which jobs you need to choose in which order to maximize your profit?

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- We don’t have any customer who will come at 1300 to get the work. So, we have 3 times.
  - 0900 - 1000
  - 1000 - 1100
  - 1100 - 1200
- These are the time slots.
- What is the strategy?
  - We are going to select the max profit.
  - Max profit is Job 1. The deadline is 2 → They will come at 11.
- Where should I put this?
  - Since they will come at 1100, I don’t need to do it between 09-10.
  - I will put it between 10-11.

09:00 - 10:00	
10:00 - 11:00	Job 1
11:00 - 12:00	

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- Next, the highest profit is Job 2.
  - I also have 2 hours.
  - I can put it between 10-11.
    - But it is full!
    - I cannot put it between 11-12 → Because they will come at 11.
  - I will put it on the left, if I have any place.
- Add it between 09-10.

09:00 - 10:00	
10:00 - 11:00	Job 1
11:00 - 12:00	

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- Next, the highest profit is Job 2.
  - I also have 2 hours.
  - I can put it between 10-11.
    - But it is full!
    - I cannot put it between 11-12 → Because they will come at 11.
  - I will put it on the left, if I have any place.
- Add it between 09-10.

09:00 - 10:00	Job 2
10:00 - 11:00	Job 1
11:00 - 12:00	

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- Next, the highest profit is Job 3.
  - Deadline is 1.
  - I need to put it in 9-10.
    - The slot is full.
    - I can't go right → They won't come
    - I need to go left, but I dont have any more spaces left.
    - I discard this job.
- Next, Job 4.
  - Deadline is 3.
  - Good, I can put it between 11-12.
  - It is empty, so I put it there.

09:00 - 10:00	Job 2
10:00 - 11:00	Job 1
11:00 - 12:00	

Job Sequencing with Deadlines

	Job 1	Job 2	Job 3	Job 4	Job 5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

- Next, the highest profit is Job 3.
  - Deadline is 1.
  - I need to put it in 9-10.
    - The slot is full.
    - I can't go right → They won't come
    - I need to go left, but I dont have any more spaces left.
    - I discard this job.
- Next, Job 4.
  - Deadline is 3.
  - Good, I can put it between 11-12.
  - It is empty, so I put it there.
- I now will do Job 2, Job 1 and Job 4 → Will have 20 + 15 + 5 = 40 profit.

09:00 - 10:00	Job 2
10:00 - 11:00	Job 1
11:00 - 12:00	Job 4

Optimal Merge Pattern

- Imagine there are two lists A and B.
  - These are sorted.
  - A: 3,8,12,10
  - B: 5,9,11,16
- How do we combine them into C ?
  - We compare step by step and take the minimum.
    - min (A[i], B[j])
      - i++ or j++
- We will have i + j steps.
  - Can we make it less?

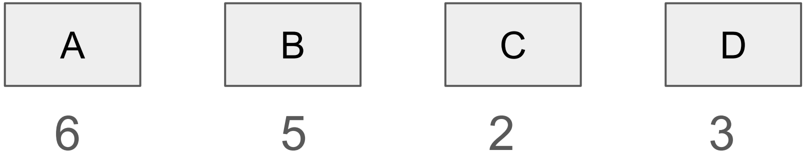
Optimal Merge Pattern

- Imagine that we have multiple lists.
  - We only have their sizes.
- Data
  - Lists → A, B, C, D
  - Sizes → 6, 5, 2, 3
- How do we combine (**merge**) them into a single list?

Example

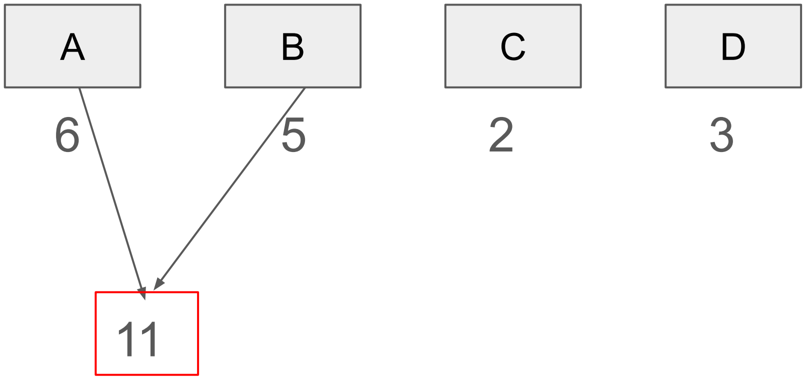
A	B	C	D
6	5	2	3

Example



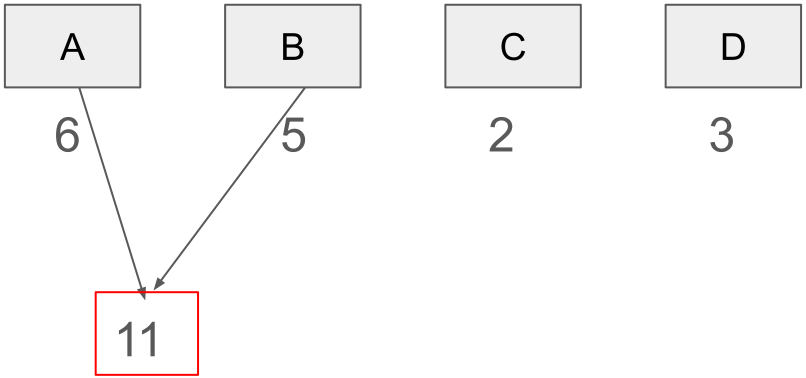
We can first merge A and B

Example



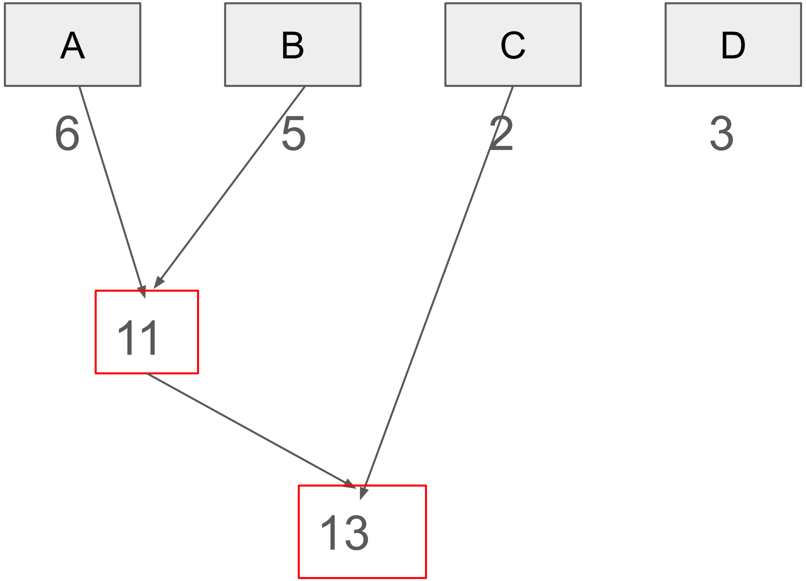
We can first merge A and B

Example



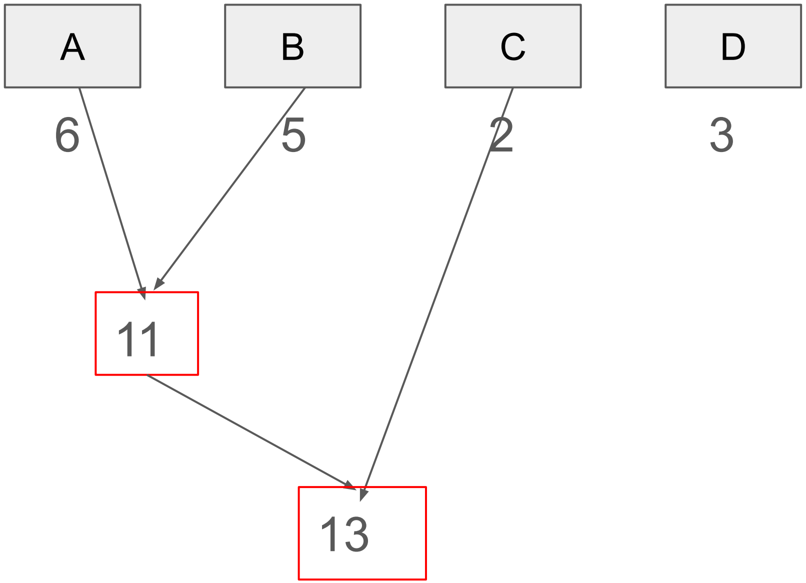
We can merge 11 with C.

Example



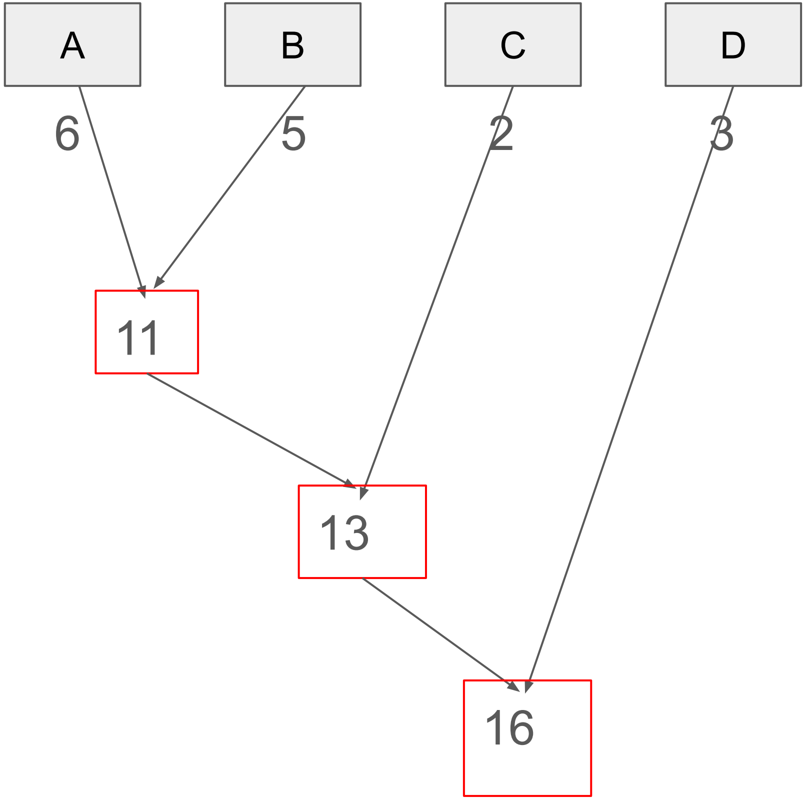
We can merge 11 with C.

Example



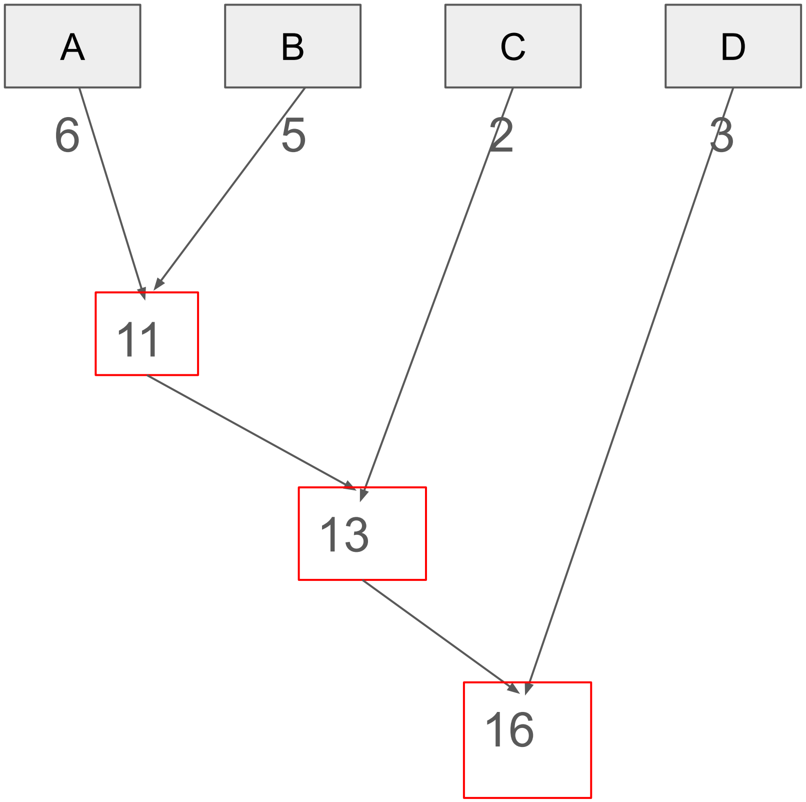
We merge it with D and done.

Example



We merge it with D and done.

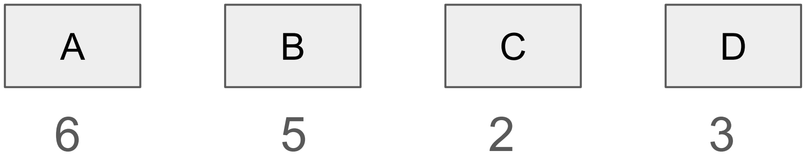
Example



11 steps
13 steps
16 steps
—----------
40 steps

can we make it
in less steps?

Example

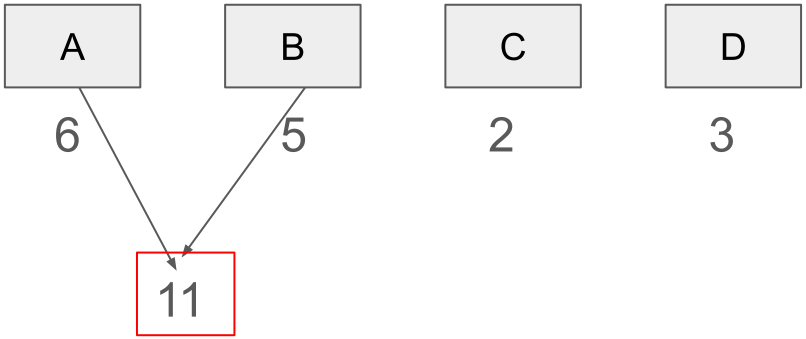


Lets try to merge A, B

Then C,D

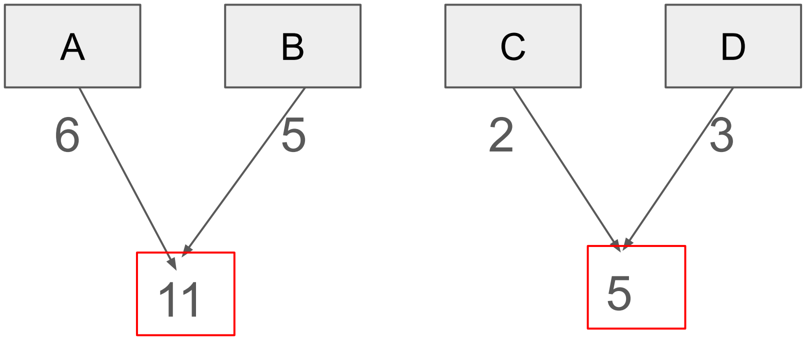
Merge them!

Example



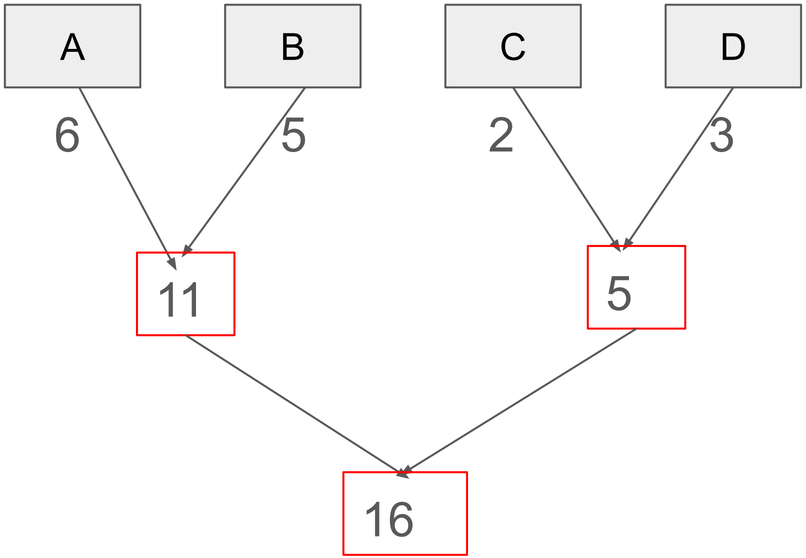
Merging A and B

Example



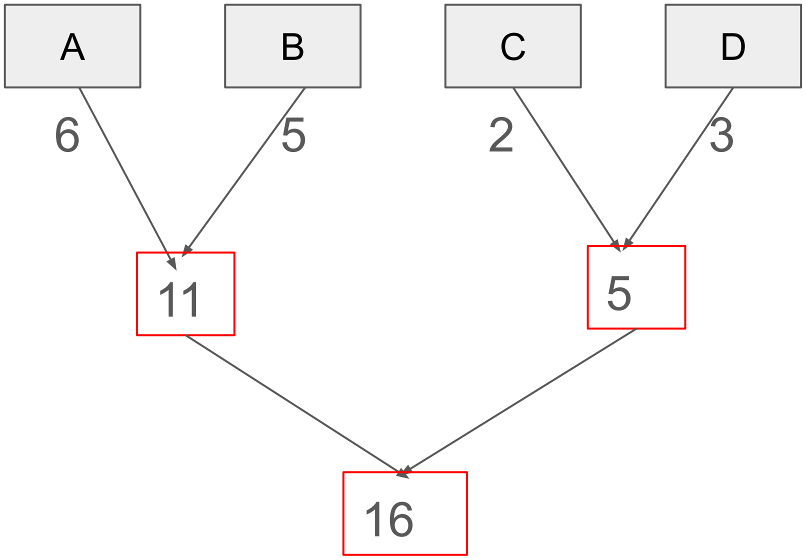
Merging C and D

Example



Merging both

Example



11 steps
5 steps
16 steps
32 steps!

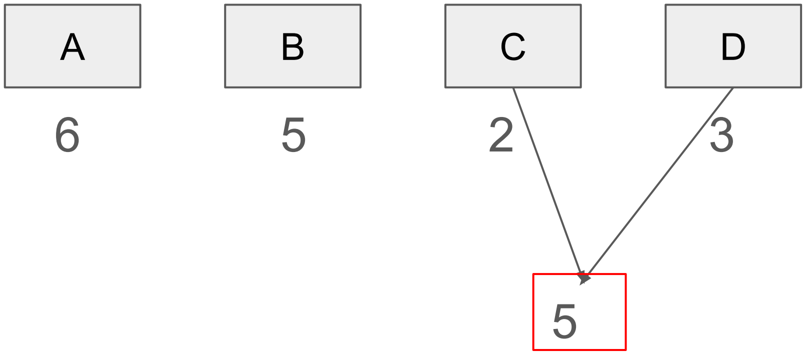
can we do in less?

Example

A	B	C	D
6	5	2	3

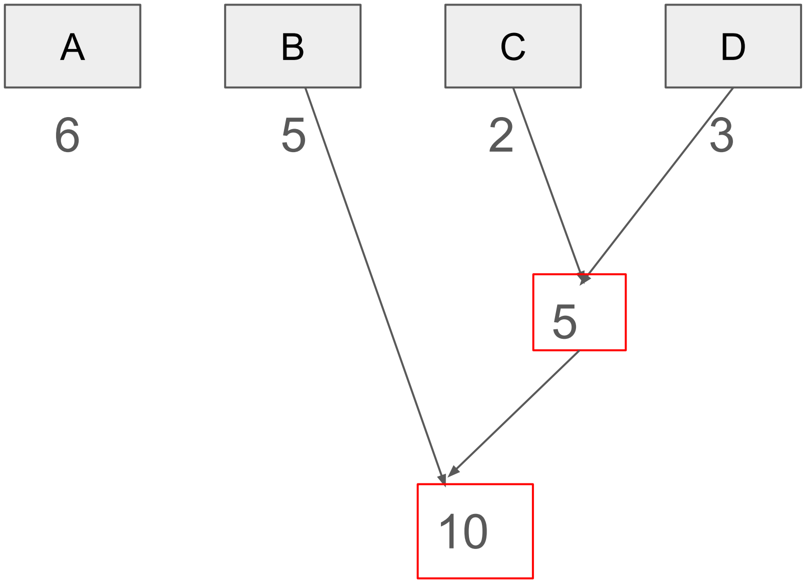
Lets start merging from the left now.

Example



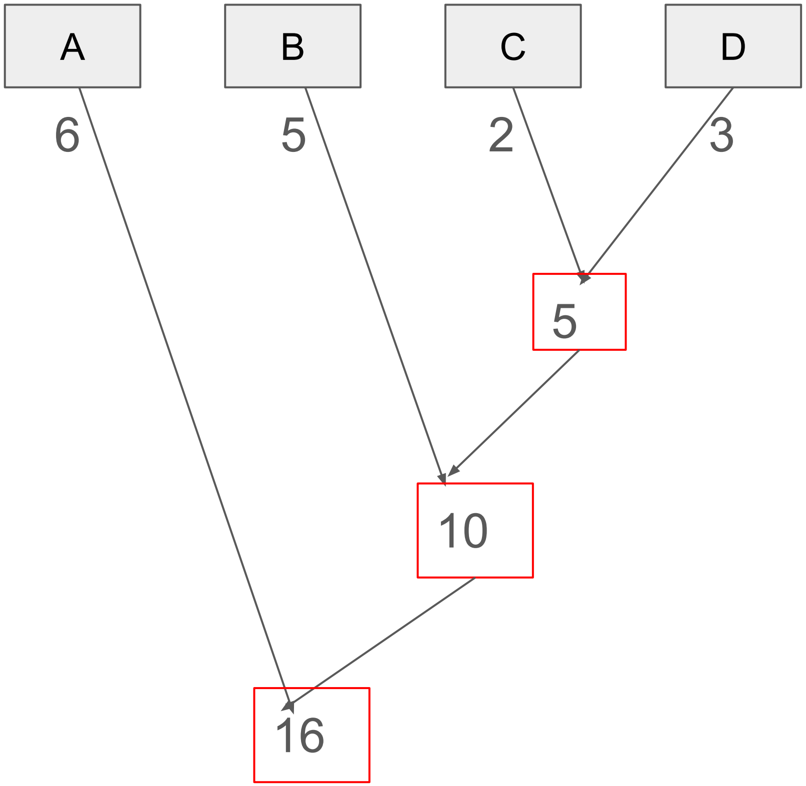
Lets start merging from the left now.

Example



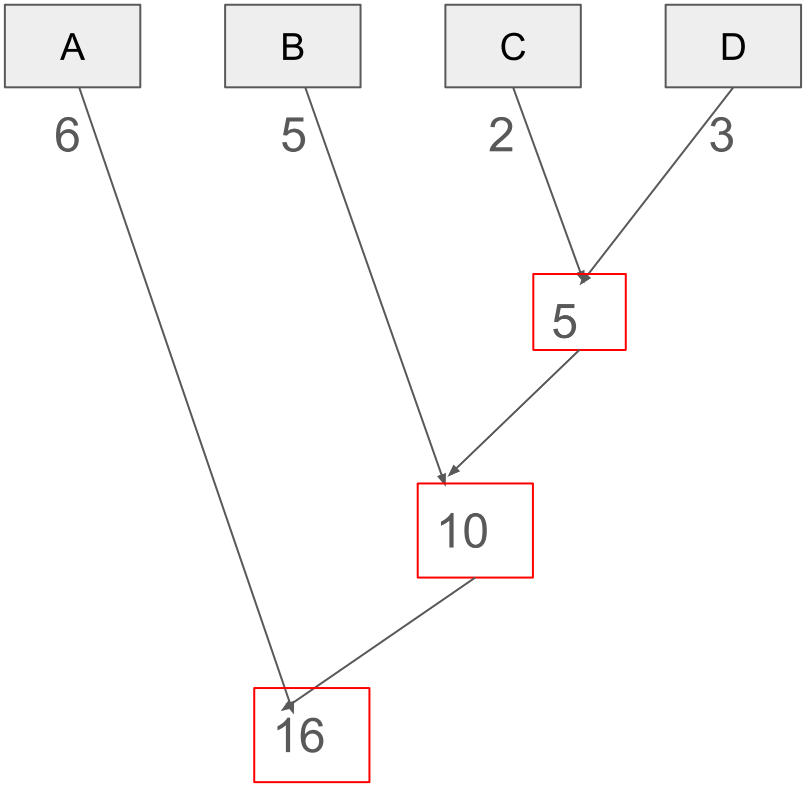
carry on with B.

Example



carry on with A

Example



5 steps
10 steps
16 steps

31 steps!

Optimal Merging

- So, we actually found a strategy.
  - If we start merging those with **smallest** sizes, we get the minimum number of steps.
- This will come handy in **Huffman Coding**

Example

	A	B	C	D	E
num of elm.	20	30	10	5	30

Example

	A	B	C	D	E
num of elm. →	20	30	10	5	30

we want to minimize the merging steps

Example

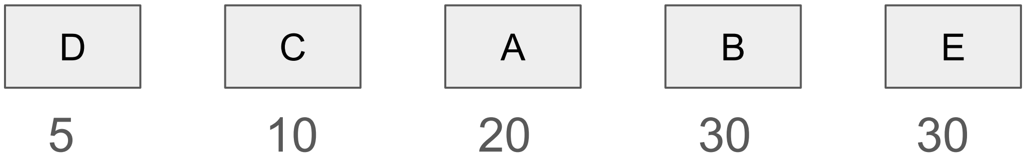
	A	B	C	D	E
num of elm.	20	30	10	5	30

A	B	C	D	E
20	30	10	5	30

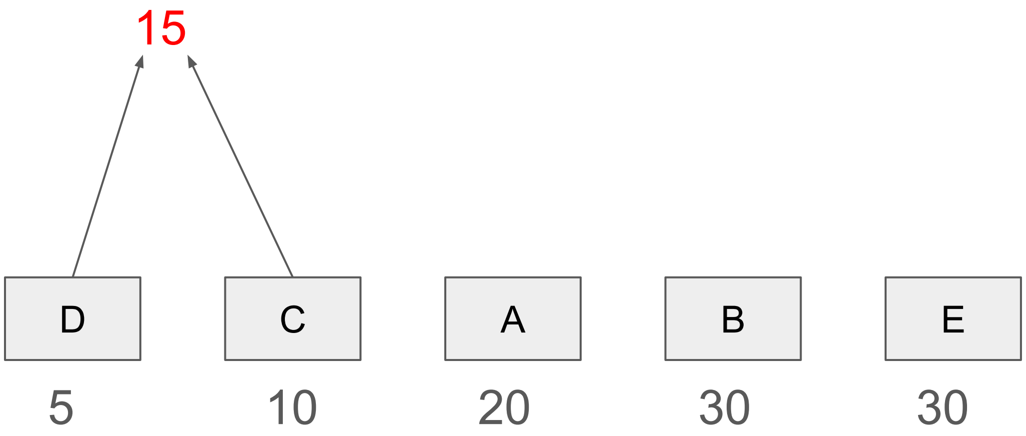
- We saw that we should start with lists having the least number of elements.
    - We will reorder them in that order.
- D - C - A - B - E

D	C	A	B	E
5	10	20	30	30

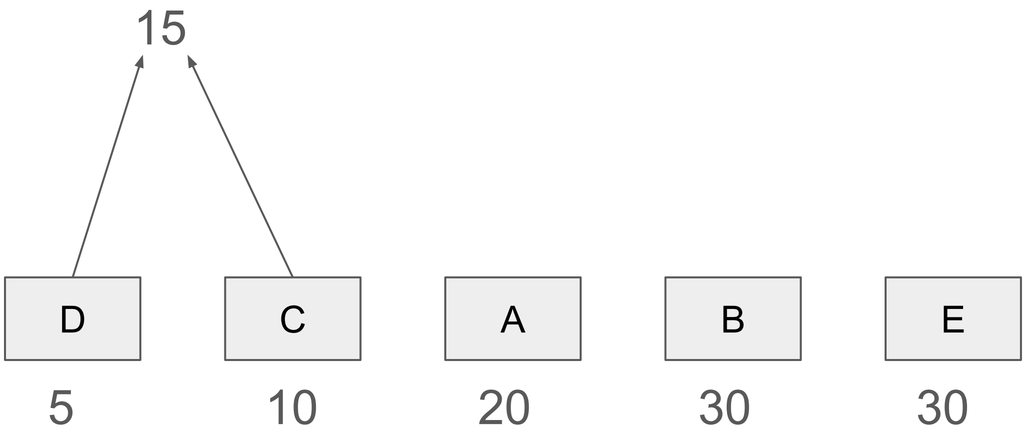
Merge D and C



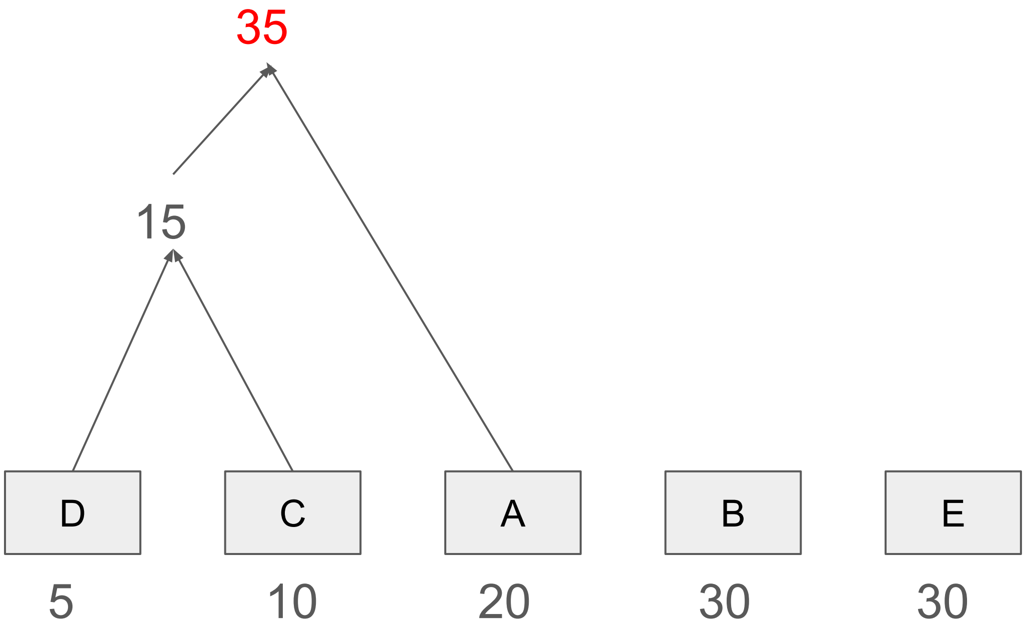
Merge D and C



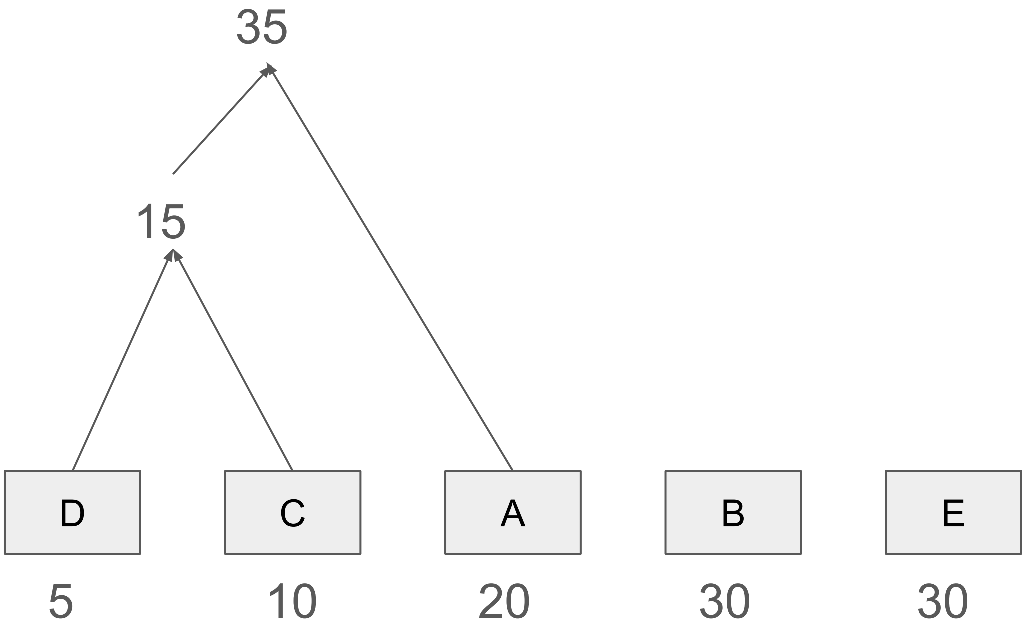
Merge with A



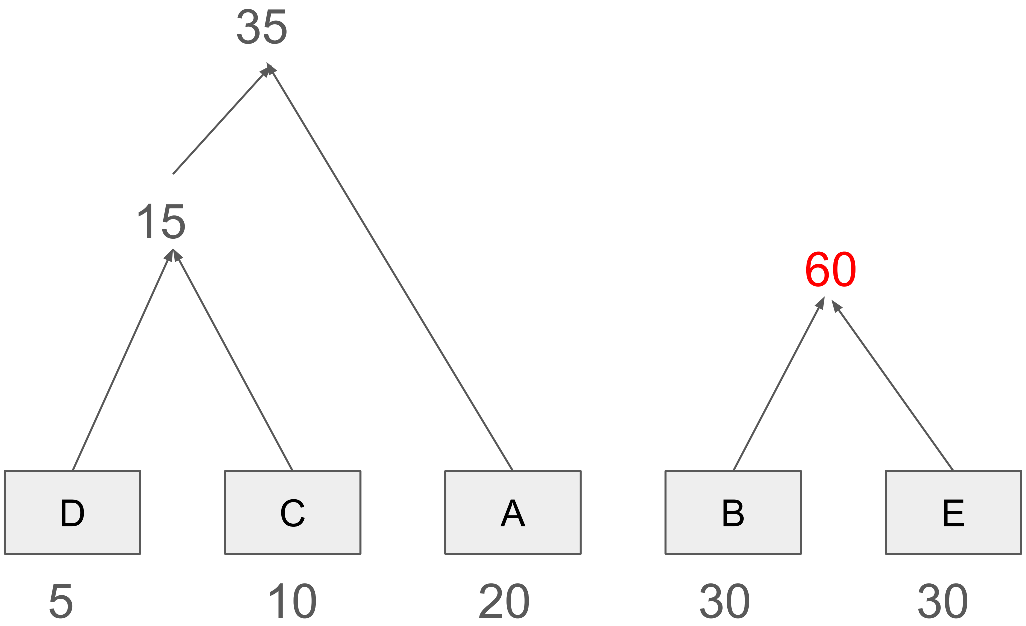
Merge with A



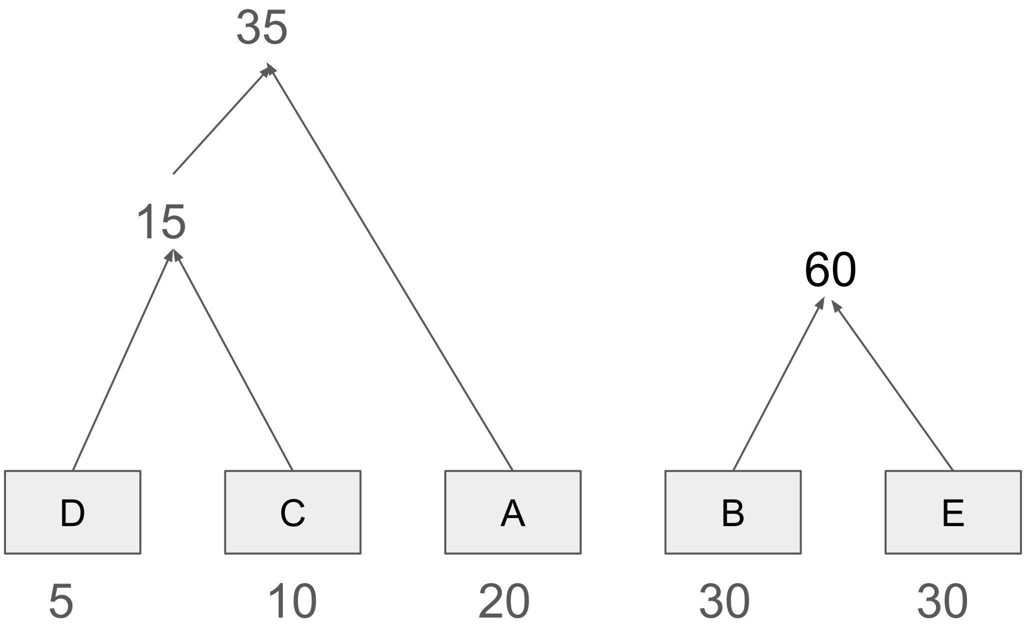
Merge B and E (smaller)



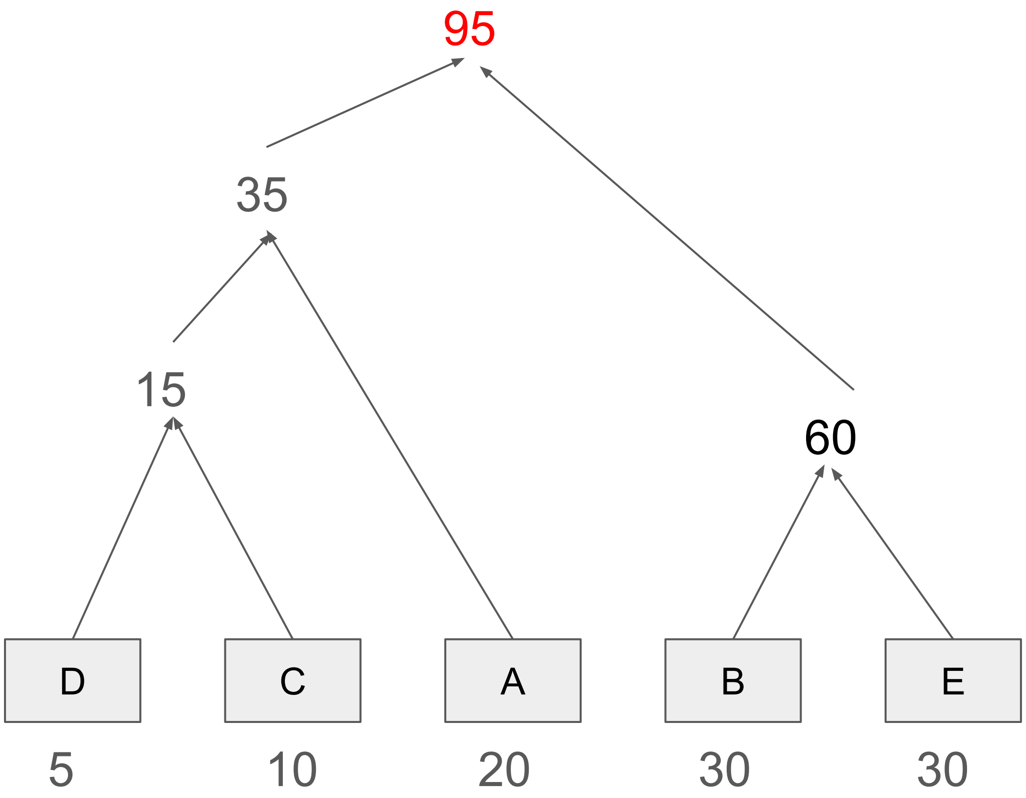
Merge B and E (smaller)



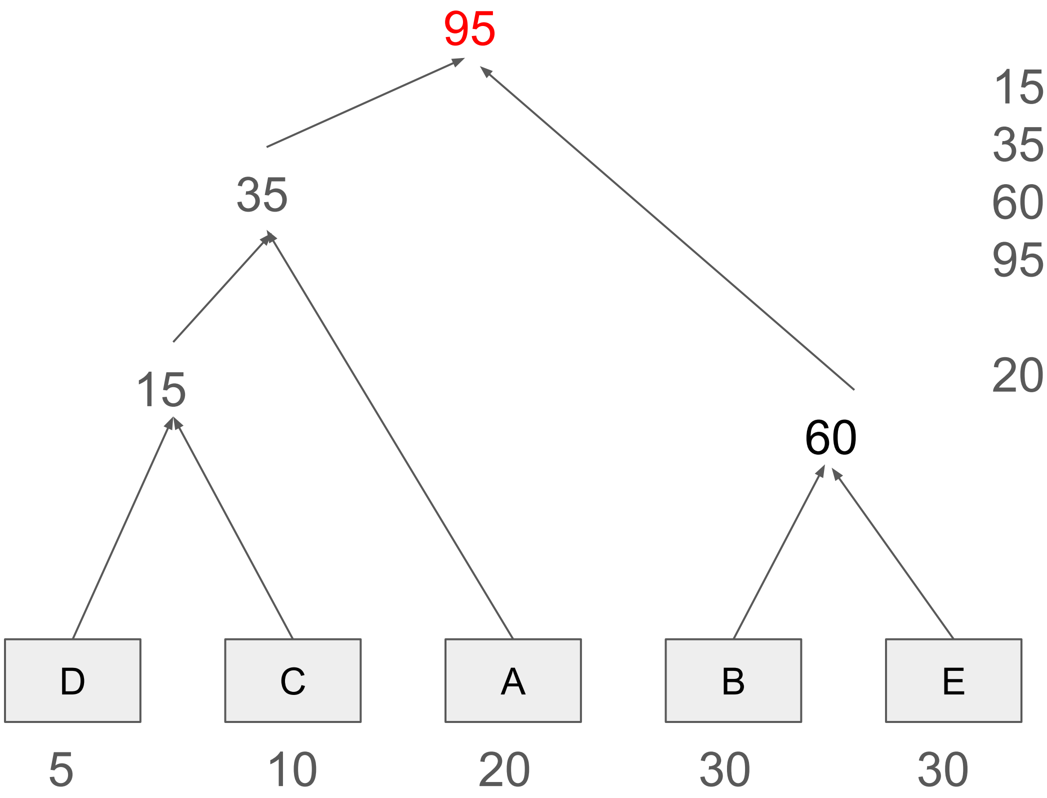
Merge remaining



Merge remaining



Merge remaining



Num of steps

15 steps
35 steps
60 steps
95 steps

205 steps

Huffman Coding

- Before huffman coding, lets talk about encoding messages.
- Previously, we said that each symbol can be represented with ASCII.
    - ASCII is 8 bits.
    - 1 bit is extra → 7 bits → 2^7 → can represent 128 symbols.
- _ bit → can be 0 or 1 → 2 symbols (2^1)
- _ _ bit → can be 00, 01, 10, 11 → 4 symbols (2 ^ 2)

- The message is :
  - BCCABBDDAECC
  - Message size is 12 characters.
  - If we use ASCII encoding it is going to be 96 bits.
- If we want to send this message over any line, we are going to use 96 bits.
  - This is a simplification.
  - We normally use additional bits for error detection etc.
- So, can we decrease the number of bits we use here?
  - If so, how?

- Remember that we can use **a slot** to represent two symbols.
- ASCII represents 127 symbols.
  - Do we need that many here?
  - We are only sending A, B, C, D and E.
    - There are 5 symbols here!
  - How many digits we need to represent 5 digits?
    - If we use 2 digits, we can represent 4.
    - If we use 3 digits, we can represent 8.
  - Apparently, we will use 3 digits.
- So, lets define our symbols now.
  - A → 000
  - B → 001
  - C → 010
  - D → 011
  - E → 100

- Now, instead of using 8 bits per symbol, we are going to use 3 bits per symbol
  - A → 000
  - B → 001
  - C → 010
  - D → 011
  - E → 100
- We had 12 characters → 12 * 3 = 36 bits. We have a lot of decrease in size!
  - However, there is a problem.
  - The problem is, we created our own encoding. The destination doesn't know this!
  - We need to send our table.
    - They know ascii. So, we are going to send A to E in ASCII (5 * 8 = 40 bits)
    - We are going to send our own encoding → 5 * 3 = 15 bits
    - In total = 55 bits.
    - The size increased, but still it is less compared to 96.

This is called Fixed Encoding

- Now, instead of using 8 bits per symbol, we are going to use 3 bits per symbol
    - A → 000
    - B → 001
    - C → 010
    - D → 011
    - E → 100
- We had 12 characters → 12 * 3 = 36 bits. We have a lot of decrease in size!
    - However, there is a problem.
    - The problem is, we created our own encoding. The destination doesn’t know this!
    - We need to send our table.
        - They know ascii. So, we are going to send A to E in ASCII (5 * 8 = 40 bits)
        - We are going to send our own encoding → 5 * 3 = 15 bits
        - In total = 55 bits.
        - 36 + 55 = **91 bits**
        - The size increased, but still it is less compared to 96.
            - As the message length increases, the difference will be significant.

- We can also have *variable encoding*
    - Now, we will see Huffman coding.
- We actually saw that by using 3 digits, we wasted some.
    - It can still get smaller.
    - How?
- Can we give 2 digits to those symbols which are used frequently?
    - And maybe give 3 digits to those who are not that frequently used?
- Let’s try it.
    - First we need to do a frequency analysis for the symbols.

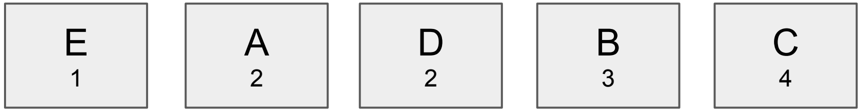
- Message
    - BCCABBDDAECC
-

Symbol	Freq.
B	
C	
C	
A	
B	
B	
D	
D	
A	
E	
C	
C	

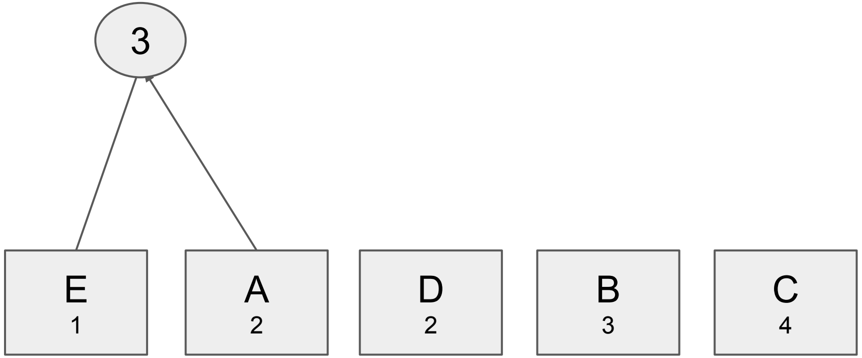
- Message
  - BCCABBDDAECC
- Now, it actually looks like a problem we solved before.
  - Optimal Merging Problem
- Let's write them in increasing order and try to merge them all. Maybe it will help!

Symbol	Freq.
B	3
C	4
A	2
D	2
E	1

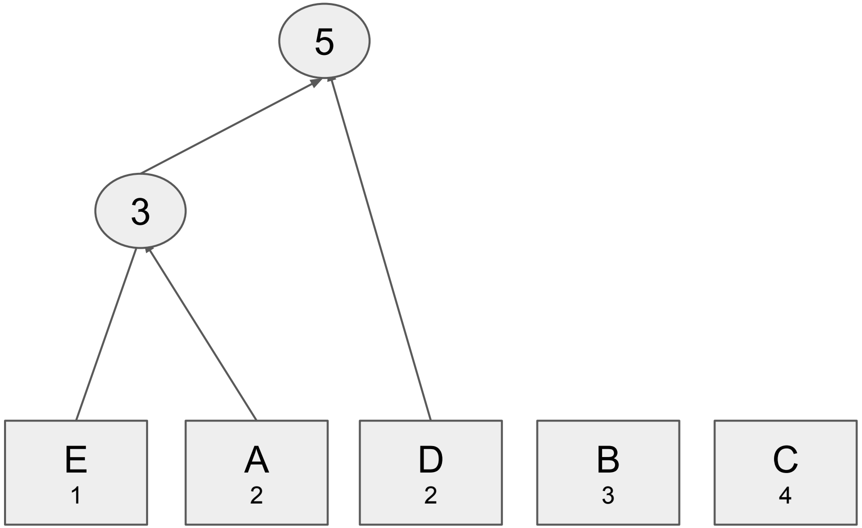
Symbol	Freq.
B	3
C	4
A	2
D	2
E	1



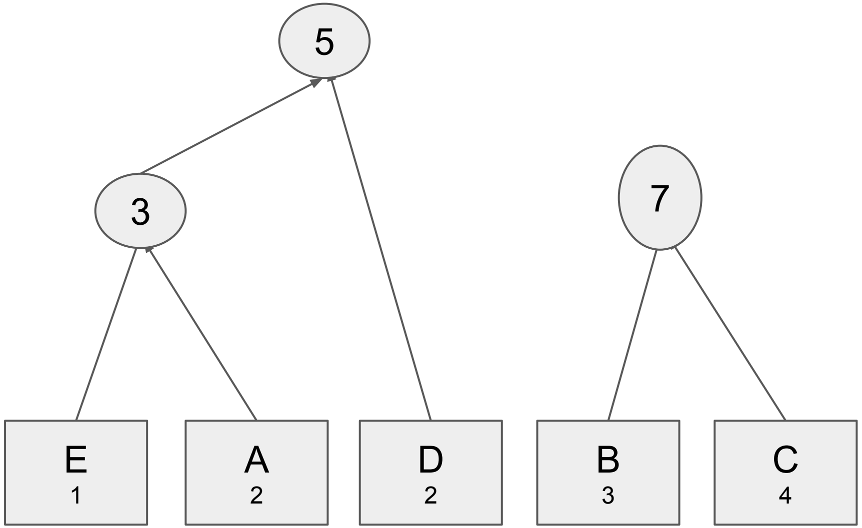
Symbol	Freq.
B	3
C	4
A	2
D	2
E	1



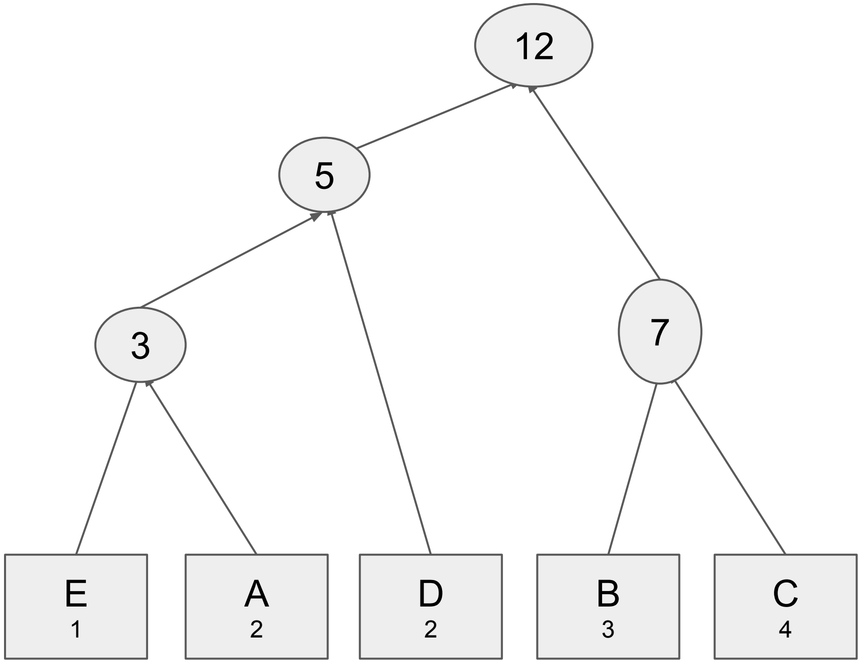
Symbol	Freq.
B	3
C	4
A	2
D	2
E	1



Symbol	Freq.
B	3
C	4
A	2
D	2
E	1

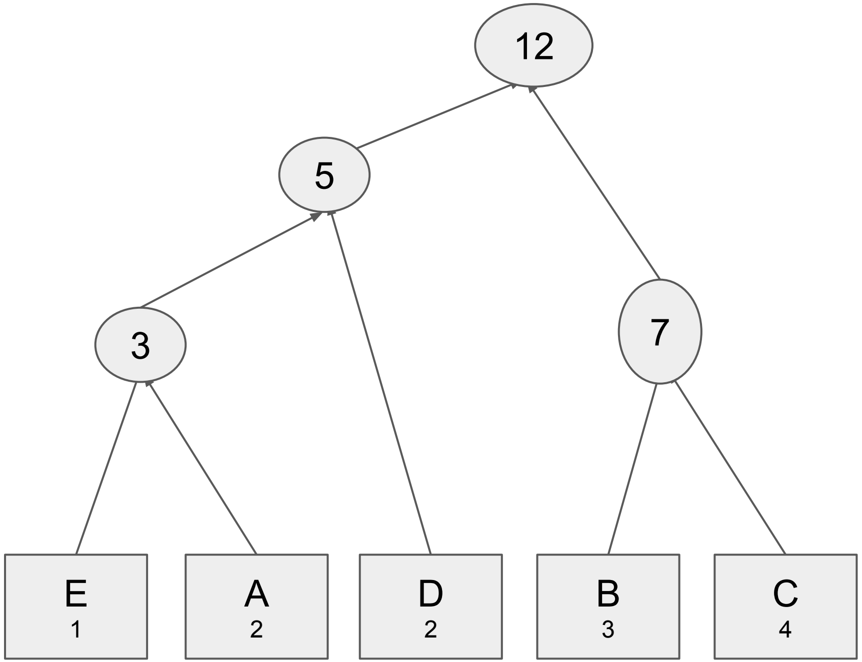


Symbol	**Freq.**
B	3
C	4
A	2
D	2
E	1



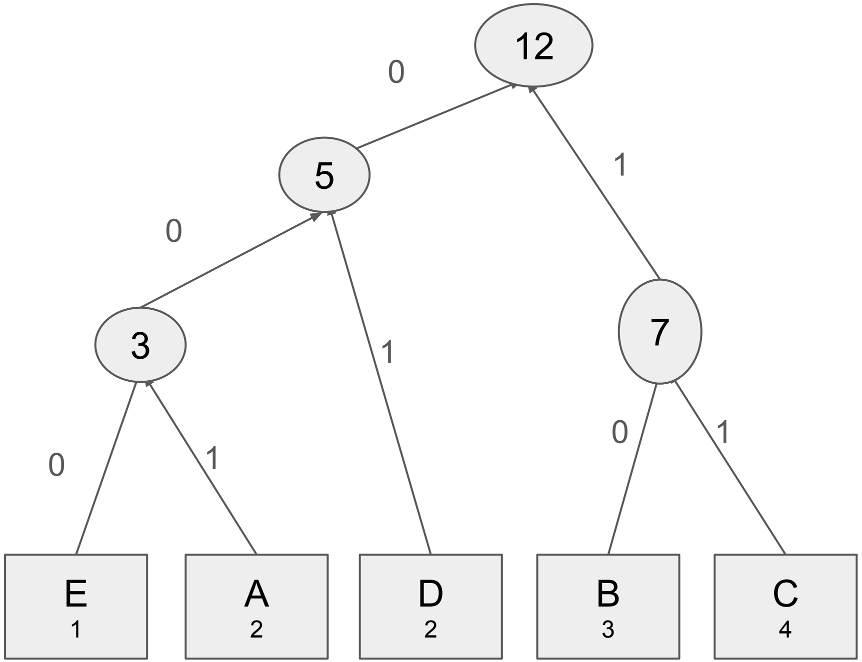
We will put 0 and 1 from left to right for all edges

Symbol	Freq.
B	3
C	4
A	2
D	2
E	1



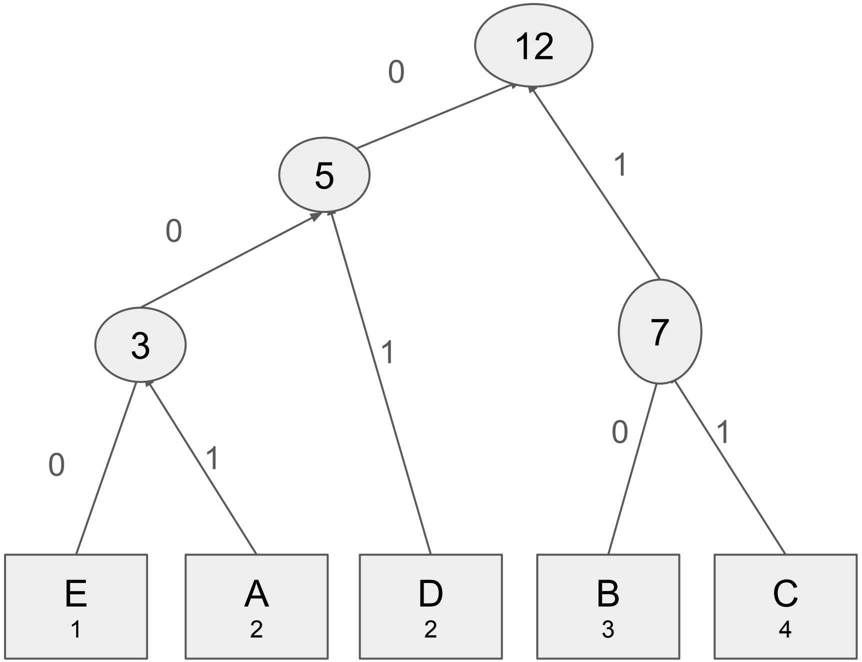
We will put 0 and 1 from left to right for all edges

Symbol	Freq.
B	3
C	4
A	2
D	2
E	1



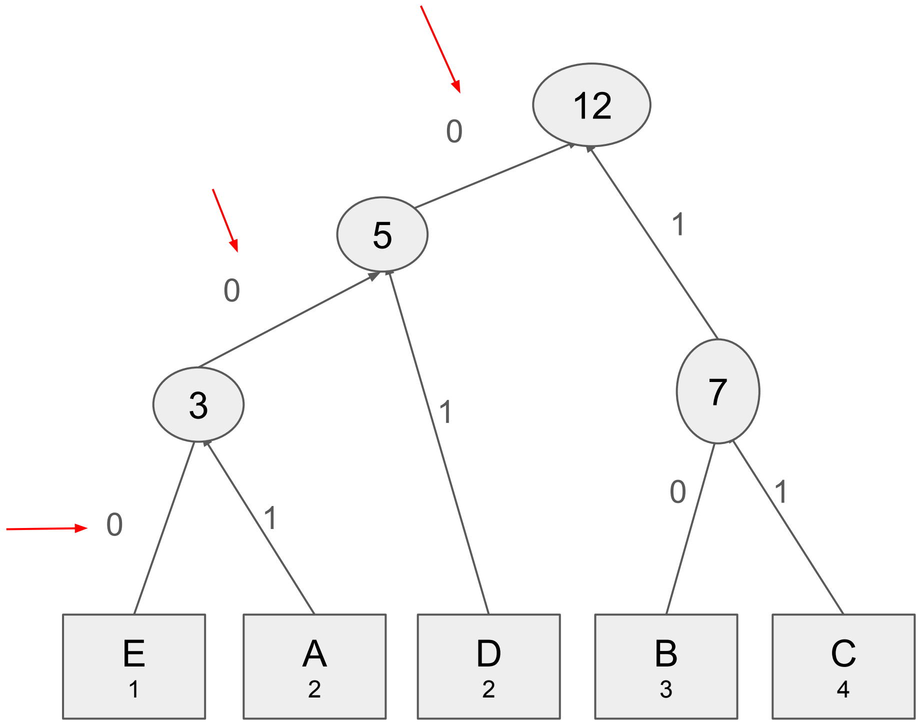
Let’s start writing codes!

Symbol	Freq.	Code
E	1	
A	2	
D	2	
B	3	
C	4	



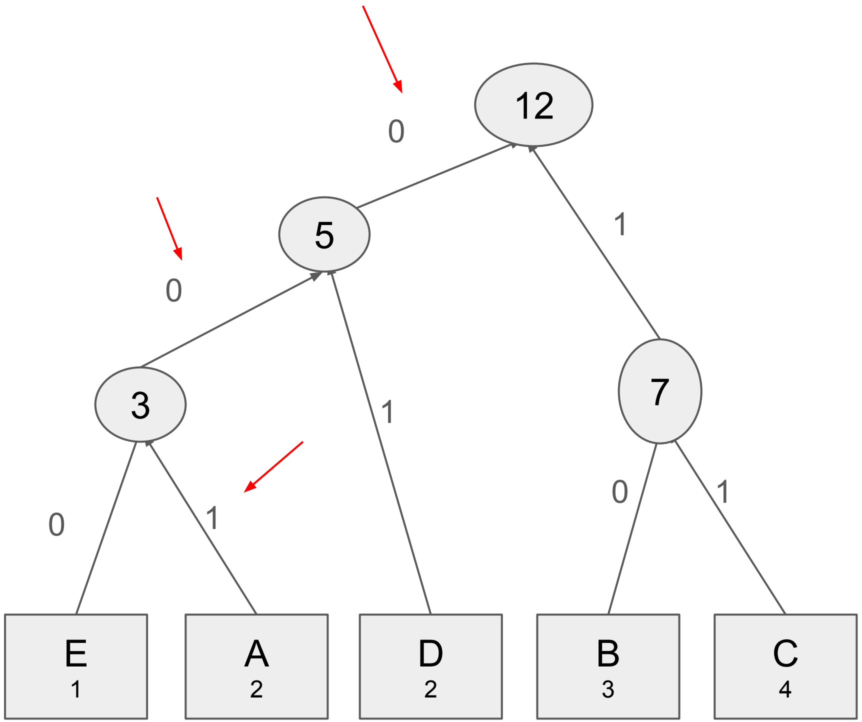
Let’s start writing codes!

Symbol	Freq.	Code
E	1	000
A	2	
D	2	
B	3	
C	4	



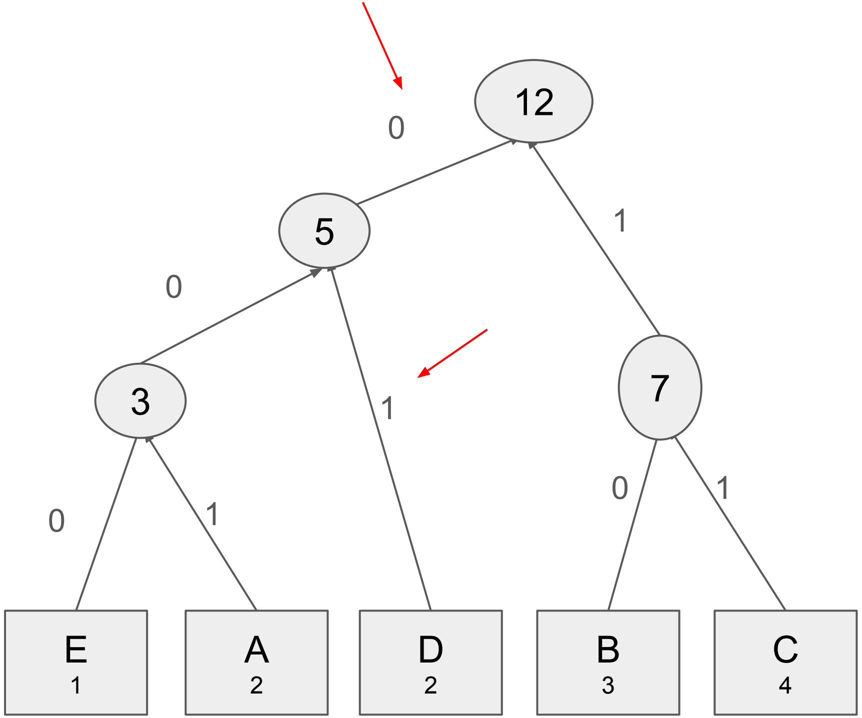
Let’s start writing codes!

Symbol	Freq.	Code
E	1	000
A	2	001
D	2	
B	3	
C	4	



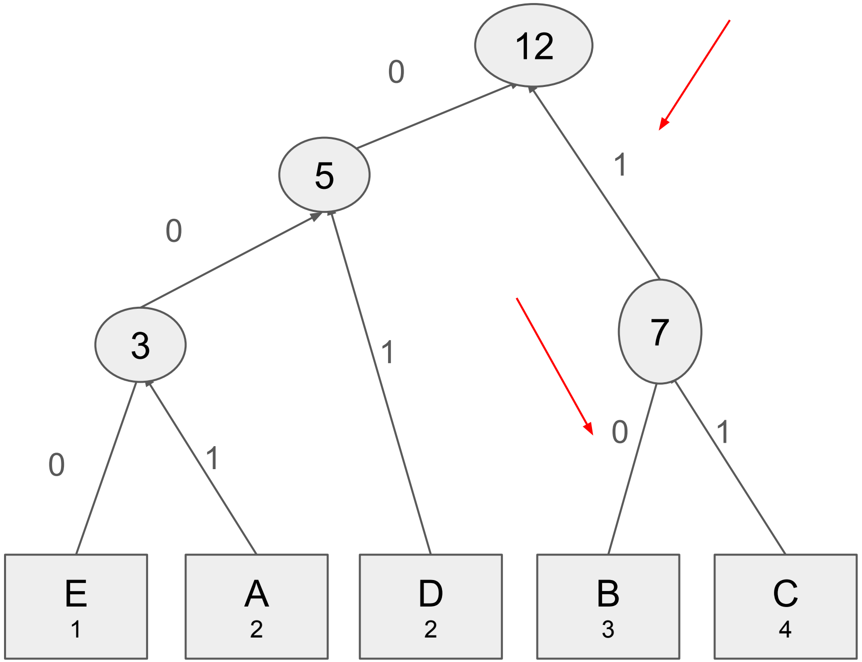
Let's start writing codes!

Symbol	Freq.	Code
E	1	000
A	2	001
D	2	01
B	3	
C	4	



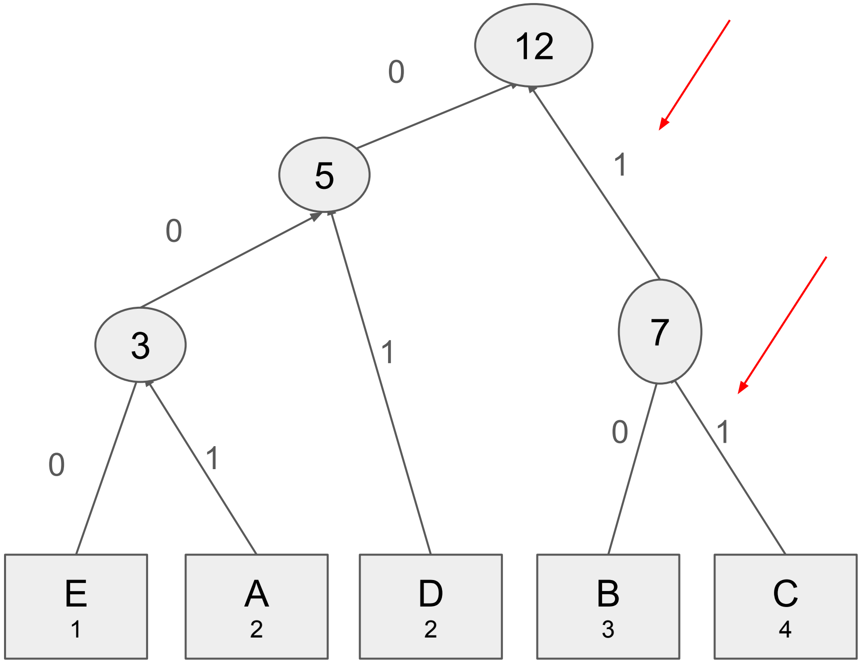
Let’s start writing codes!

Symbol	**Freq.**	**Code**
E	1	000
A	2	001
D	2	01
B	3	10
C	4	



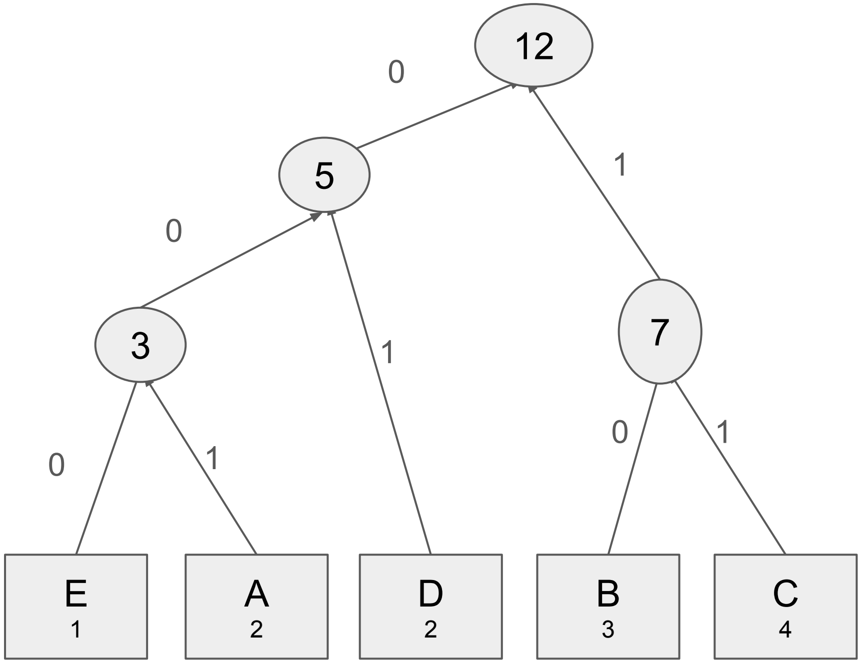
Let's start writing codes!

Symbol	Freq.	Code
E	1	000
A	2	001
D	2	01
B	3	10
C	4	11



Codes are done!

Symbol	Freq.	Code
E	1	000
A	2	001
D	2	01
B	3	10
C	4	11



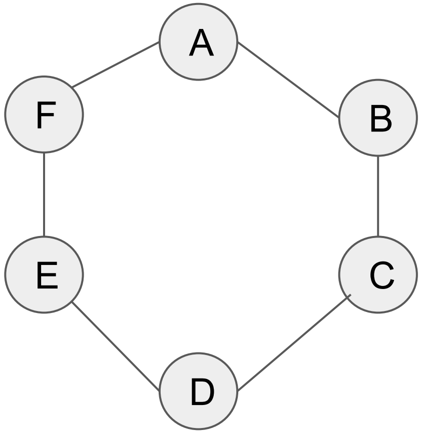
Let's calculate the message length again

- Originally, we were supposed to send 12 * 8 = 96 bits
- Now we will send
  - E for 1 time → **3 bits**
  - A for 2 times → 3 * 2 = **6 bits**
  - D for 2 times → 2 * 2 = **4 bits**
  - B for 3 times → 2 * 3 = **6 bits**
  - C for 4 times → 2 * 4 = **8 bits**
  - **Total = 27 bits**
- Let's not forget we need to send the table too!
  - 5 chars, each 8 bits → 40 bits
  - 3,3,2,2,2 → 12 bits
  - Total → 52 bits
- **27 + 52 = 77 bits in total!**

Symbol	Freq.	Code
E	1	000
A	2	001
D	2	01
B	3	10
C	4	11

Minimum Cost Spanning Tree

Spanning Tree



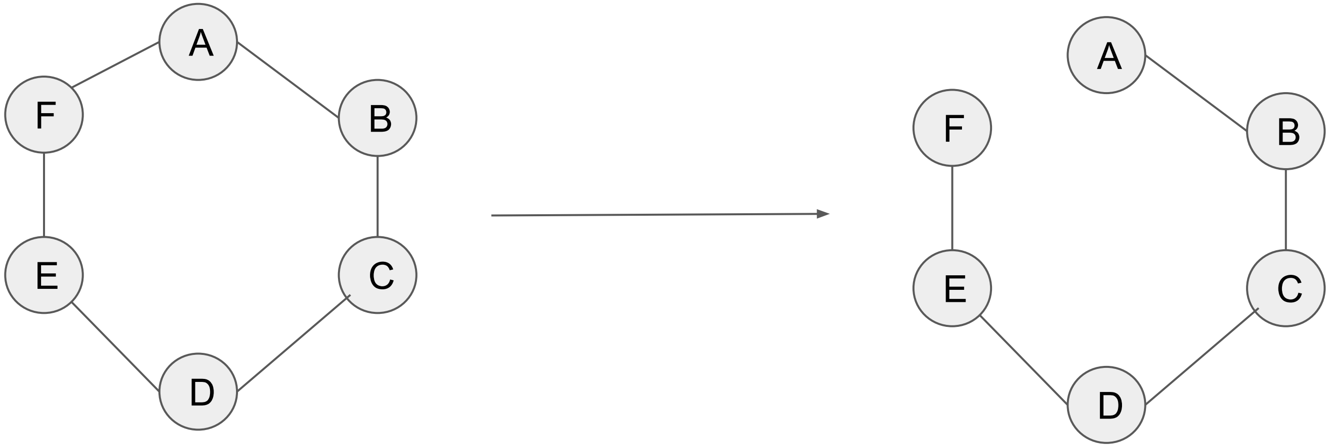
this is a graph.

G = (V,E)

V = {A,B,C,D,E,F}

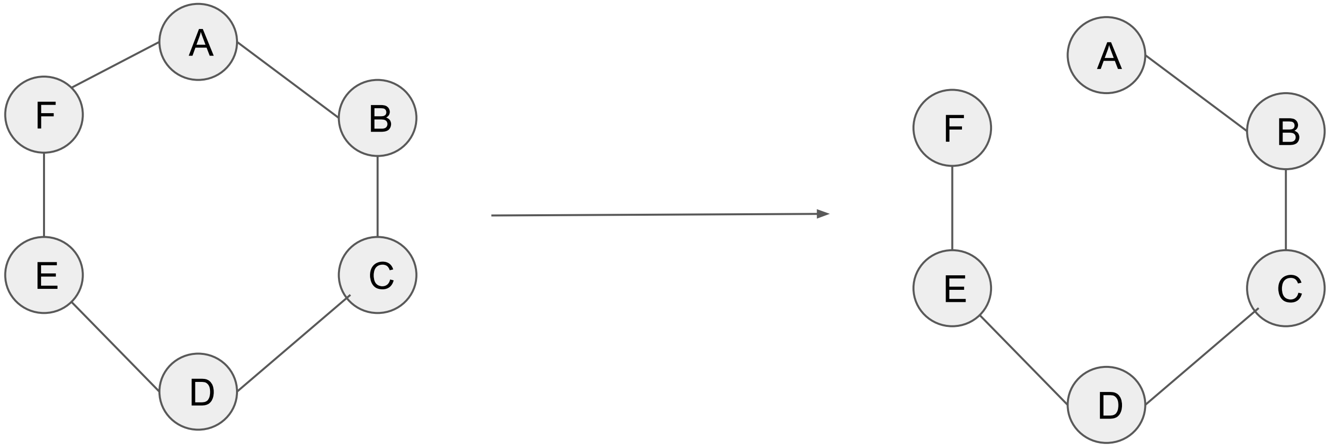
E = { (A,B), (B,C), …}

Spanning Tree



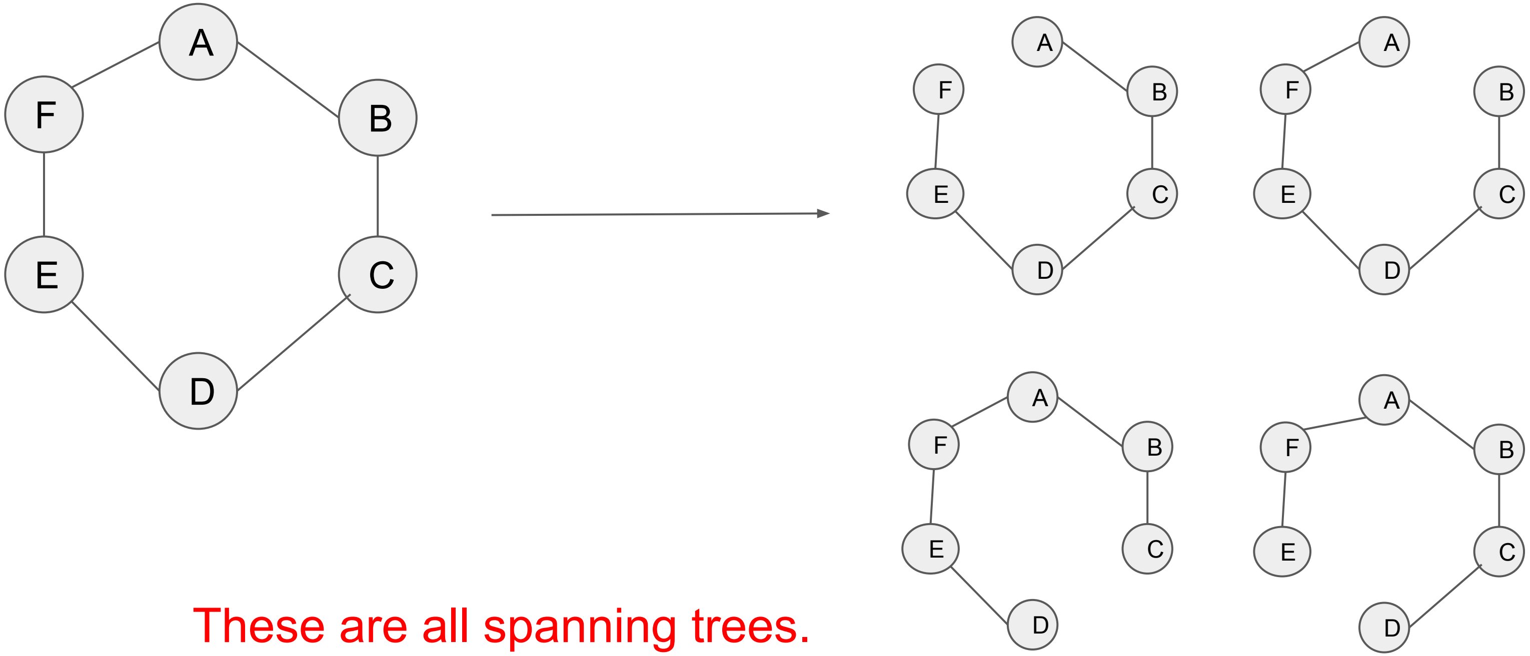
this is a spanning tree
no cycles

Spanning Tree



this is a spanning tree
we have the same
vertices, but the
number of edges are
V - 1

Spanning Tree



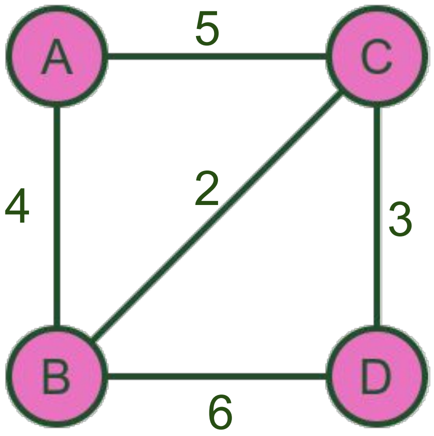
These are all spanning trees.
There are multiple.
How many are possible?

of spanning trees

- We have 6 vertices.
  - In a spanning tree, we are going to use **V-1** edges.
  - What are the possibilities?

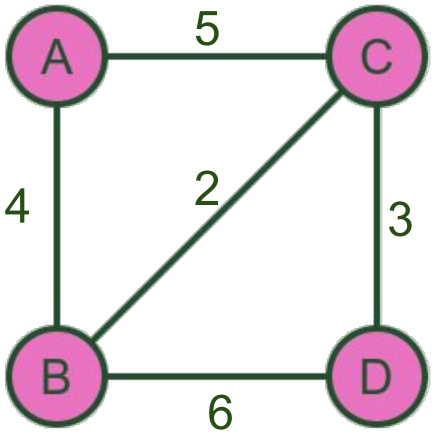
$$\binom{6}{5}$$

Let's do it with a weighted graph

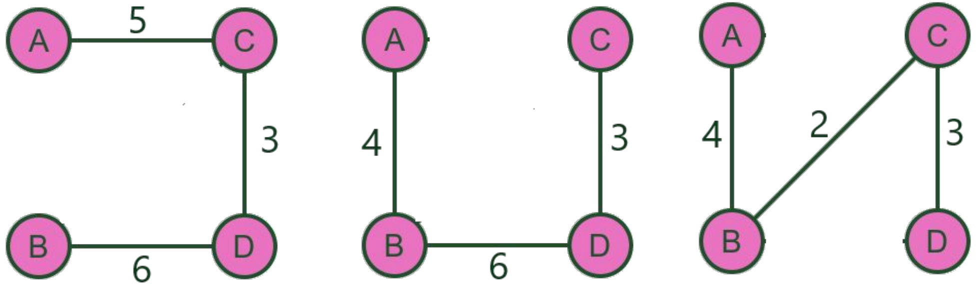


Again, we can draw spanning trees here too.

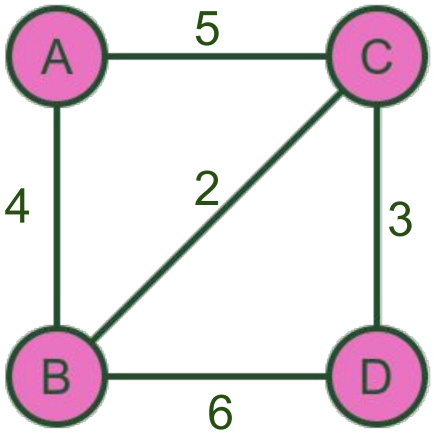
Let’s do it with a weighted graph



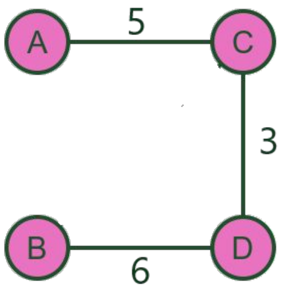
Again, we can draw spanning trees here too.



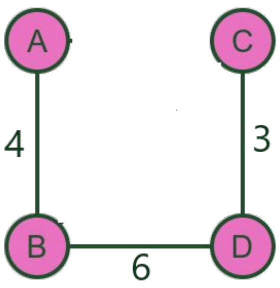
Let’s do it with a weighted graph



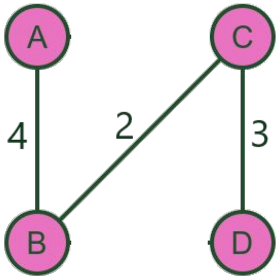
For each spanning tree there is a **cost**



cost = 14



cost = 13



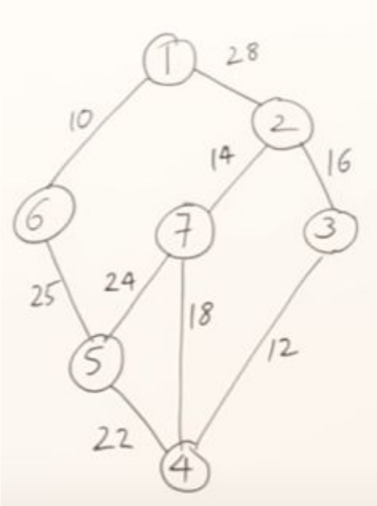
cost = 9

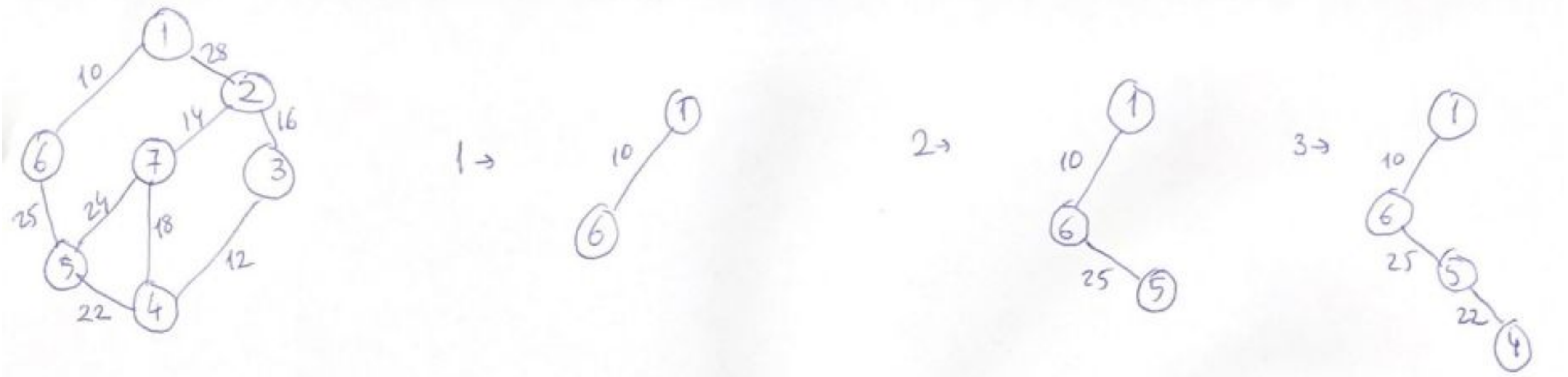
Minimum Cost Spanning Tree

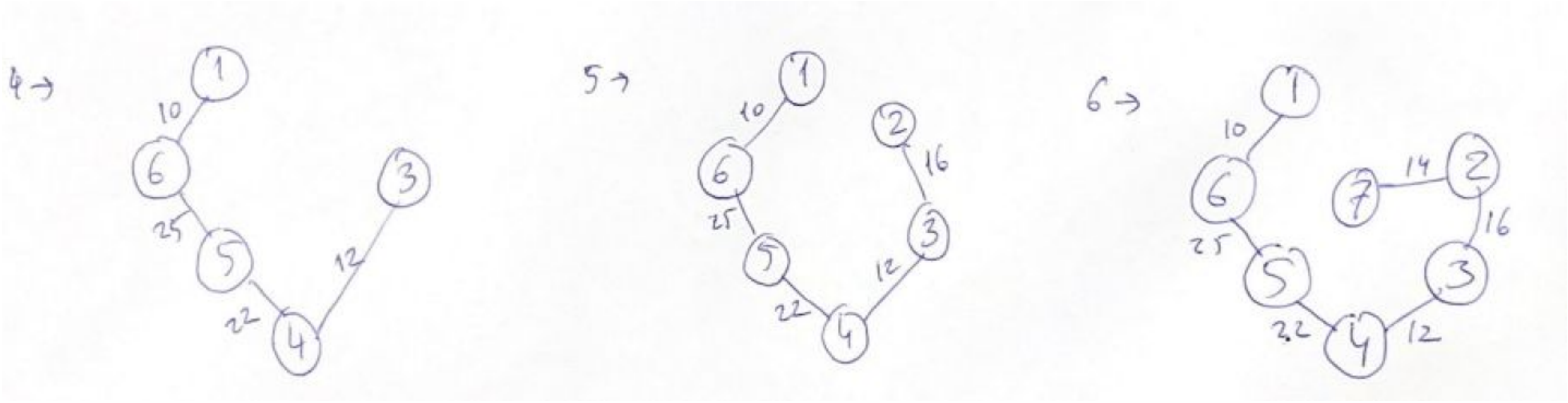
- Can we find the spanning tree with the minimum cost?
  - Yes. We can find every spanning tree and choose the one with min cost.
    - Not effective.
  - Do we have a greedy method?
    - Yes!
    - So, we can find it without creating all spanning trees.
  - 2 algorithms:
    - Prim’s Algorithm & Kruskal’s Algorithm

Prim's Algorithm

- You select minimum edge.
  - 1 to 6.
- You select the next minimum edge, but make sure that it is **connected** to previously selected edge.
  - 3,4 is the next (12)
  - But not connected to what we chose.
  - Always maintain a tree
- What are connected? 1,2 (28) and 6,5 (25)
  - Choose 6,5.
- Now, we have multiple
  - 5,7 (24) ; 5,4 (22) → choose 5,4
- We go on like this.



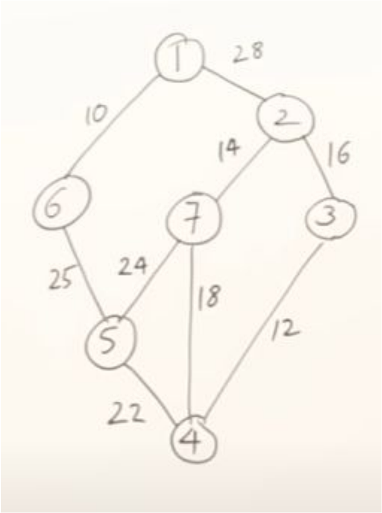

1 →
2 →
3 →

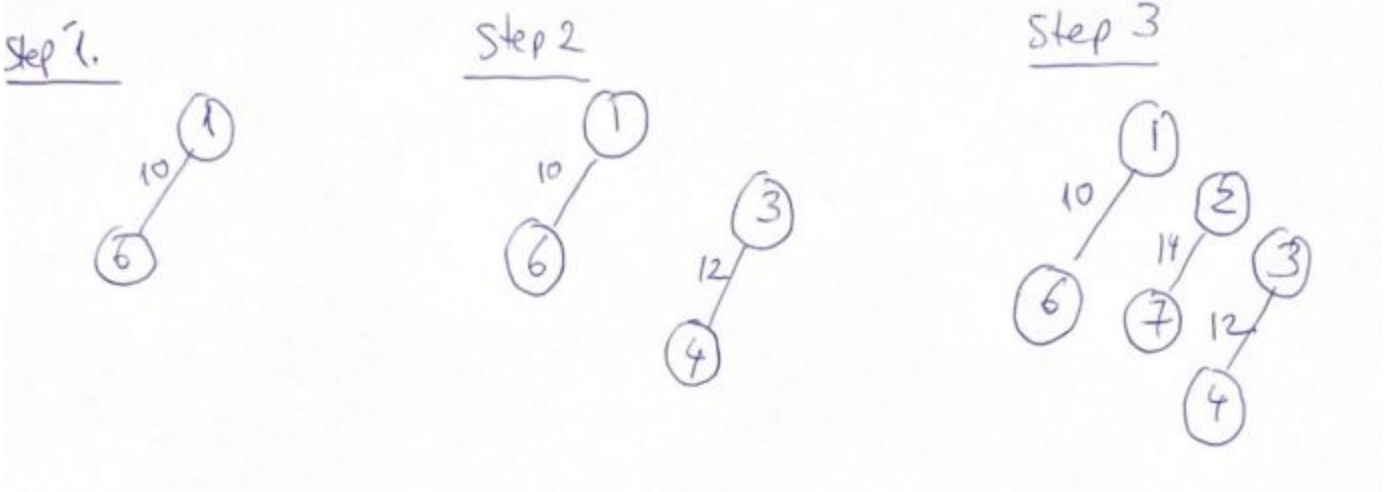


cost is 99.
sum all weights

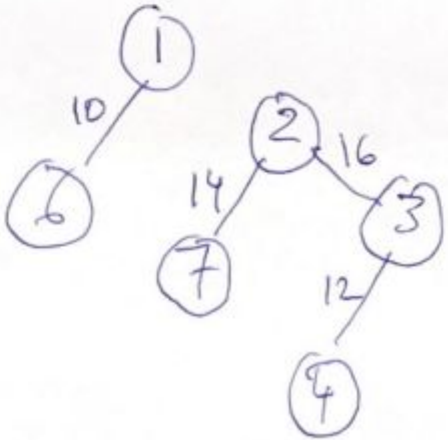
Kruskal

- Always select a minimum cost edge.
  - 1,6 → min
- Next, choose the minimum again. Now we don't need any connections like Prim.
  - 3,4 → 12. min.
- Next:
  - 2,7 → 14
  - 2,3 → 16
  - 7,4 → 18 → oops. a cycle.
- If there is a cycle, don't include it!
- So in Kruskal, we select the minimum edge cost as long as it is not forming a cycle!

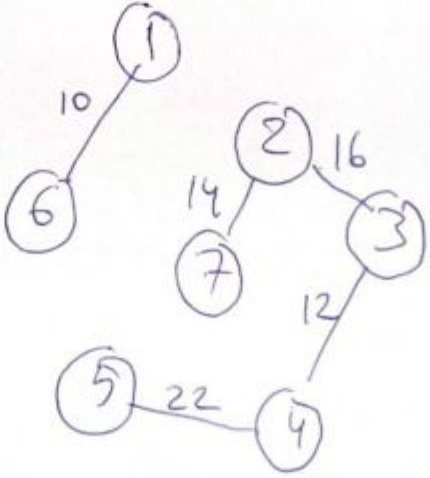



Step 1.
1
10
6
Step 2
1
10
6
3
12
4
Step 3
1
10
6
2
14
7
3
12
4

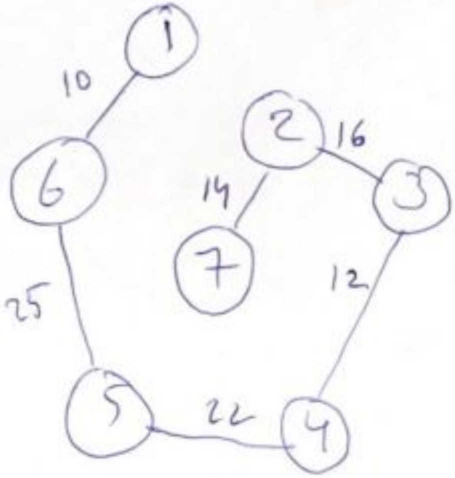
Step 4



Step 5 [can't add 7,4 bc it's a cycle]



Step 6 [7,5 can't be added · cycle]



cost = 99 //