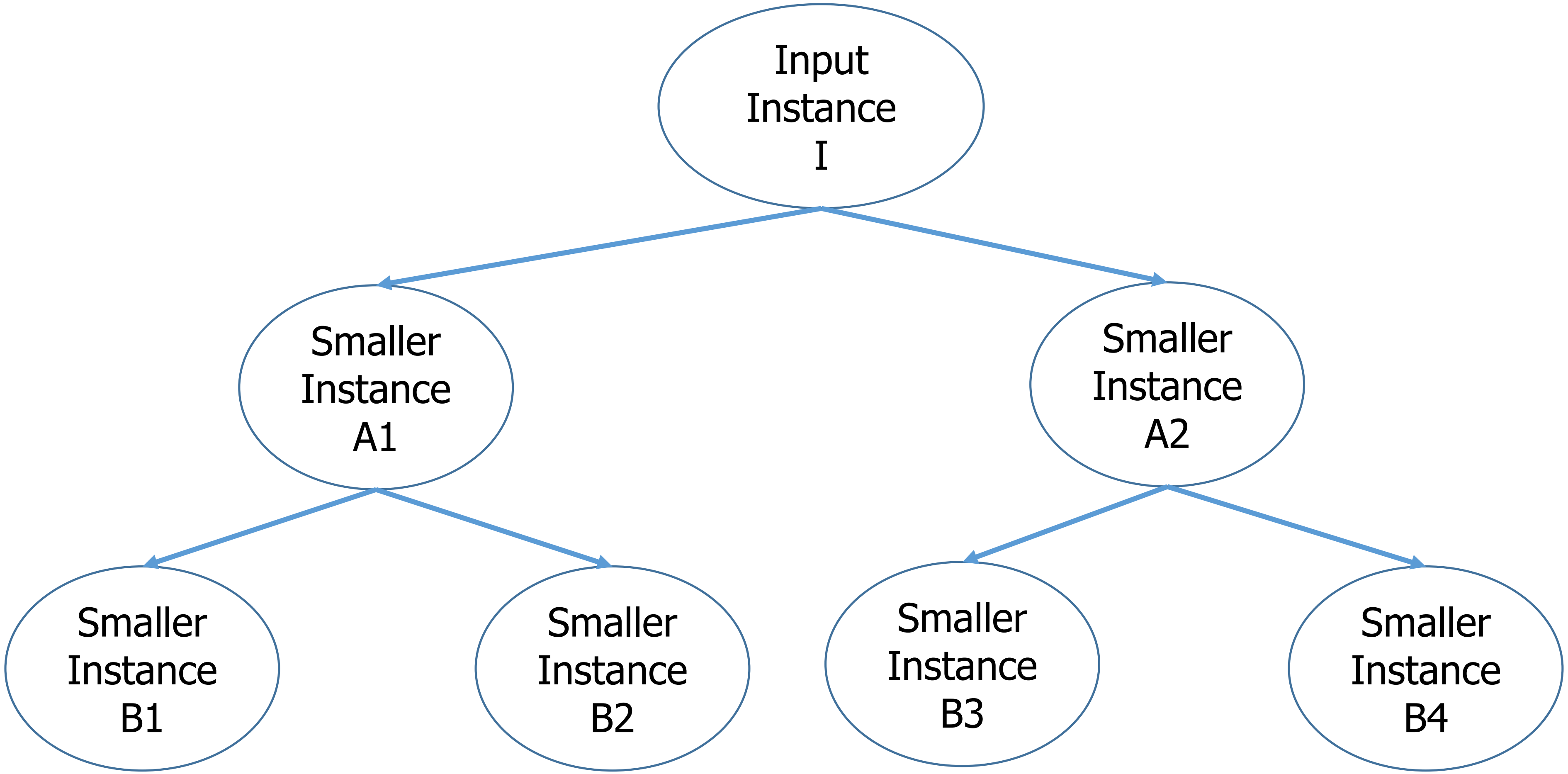
Dynamic Programming

Venkatesan Chakaravarthy
IBM Research, Bangalore

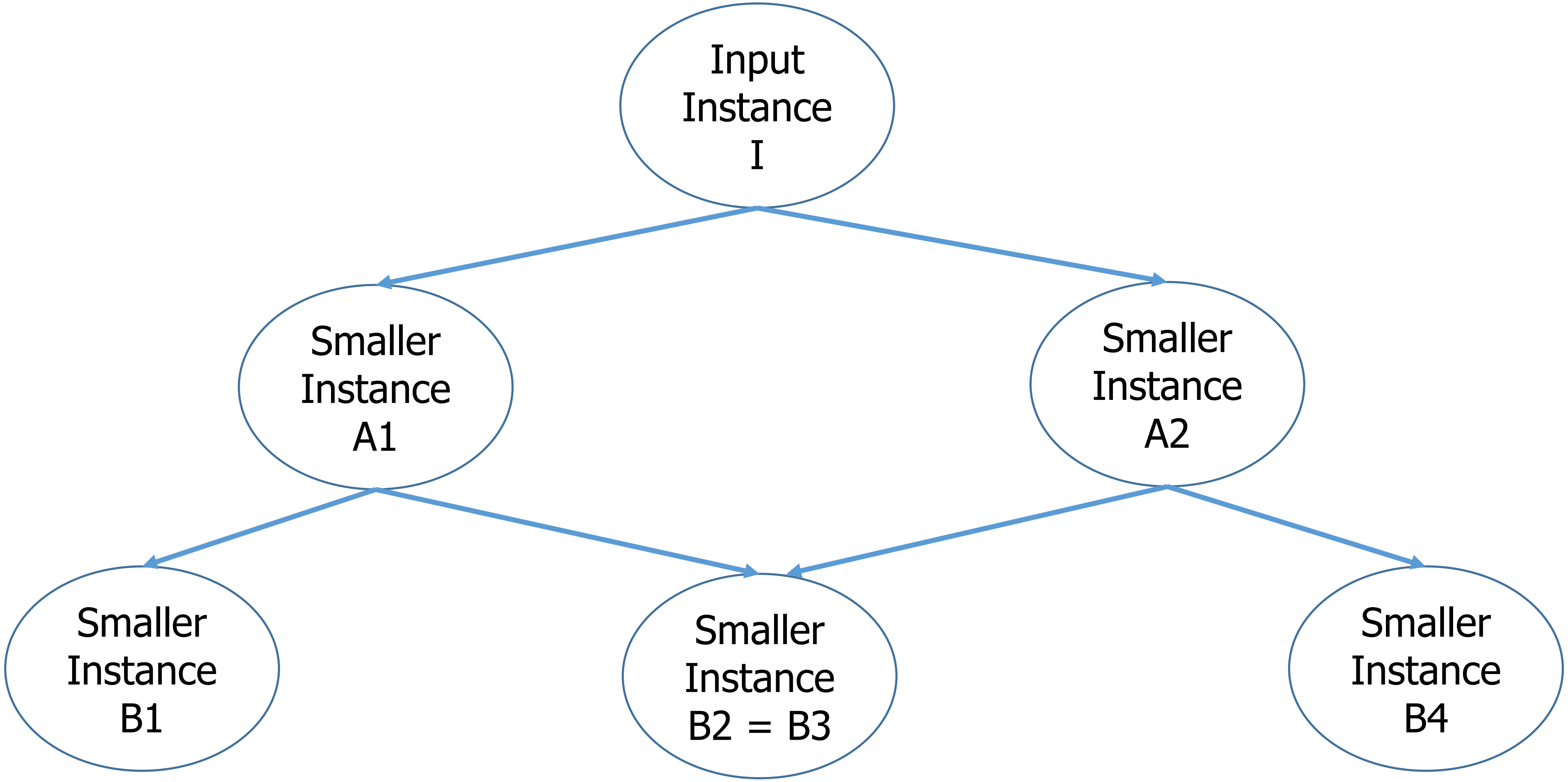
Recursion or Induction

- Solve given problem instance by reducing it to smaller instances

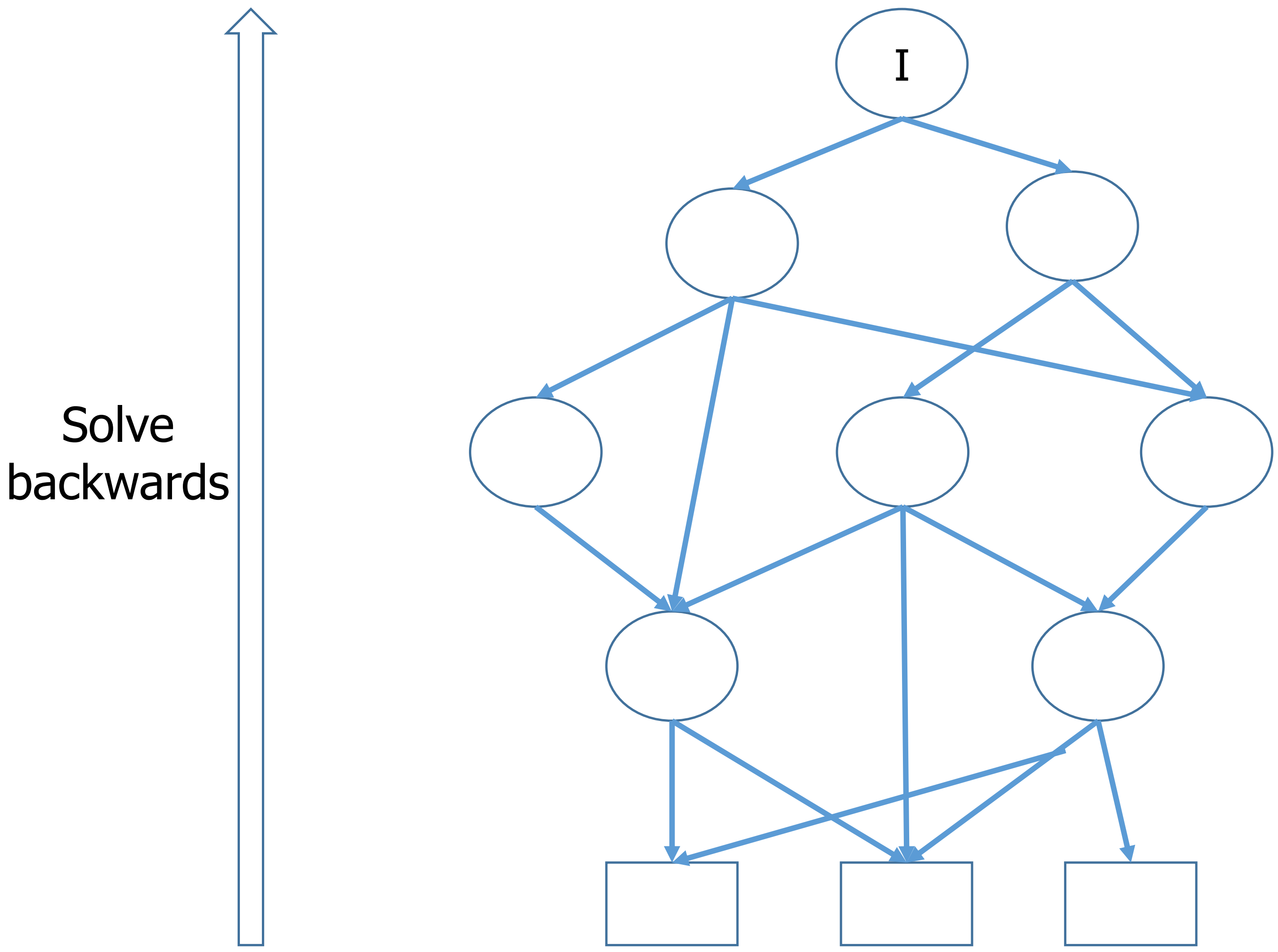


Dynamic Programming

Gospel of Dynamic Program
Do not repeat the same work: Save and reuse



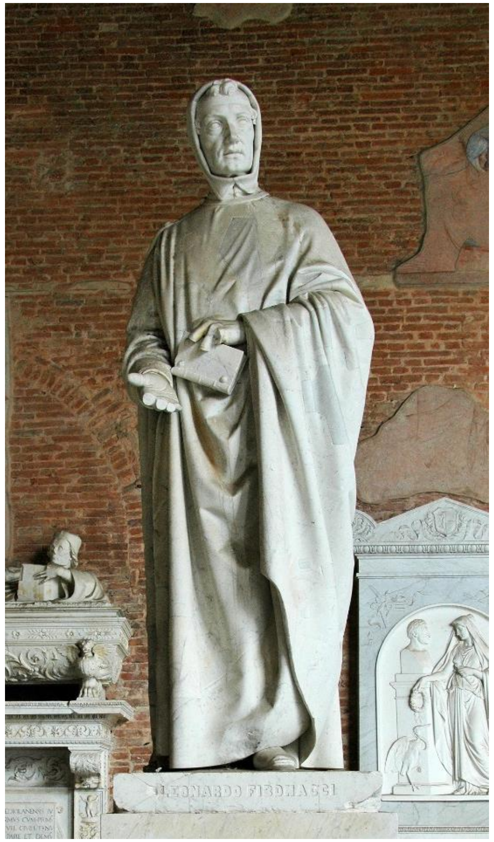
Directed Acyclic Graph (DAG) Structure



Fibonacci Sequence

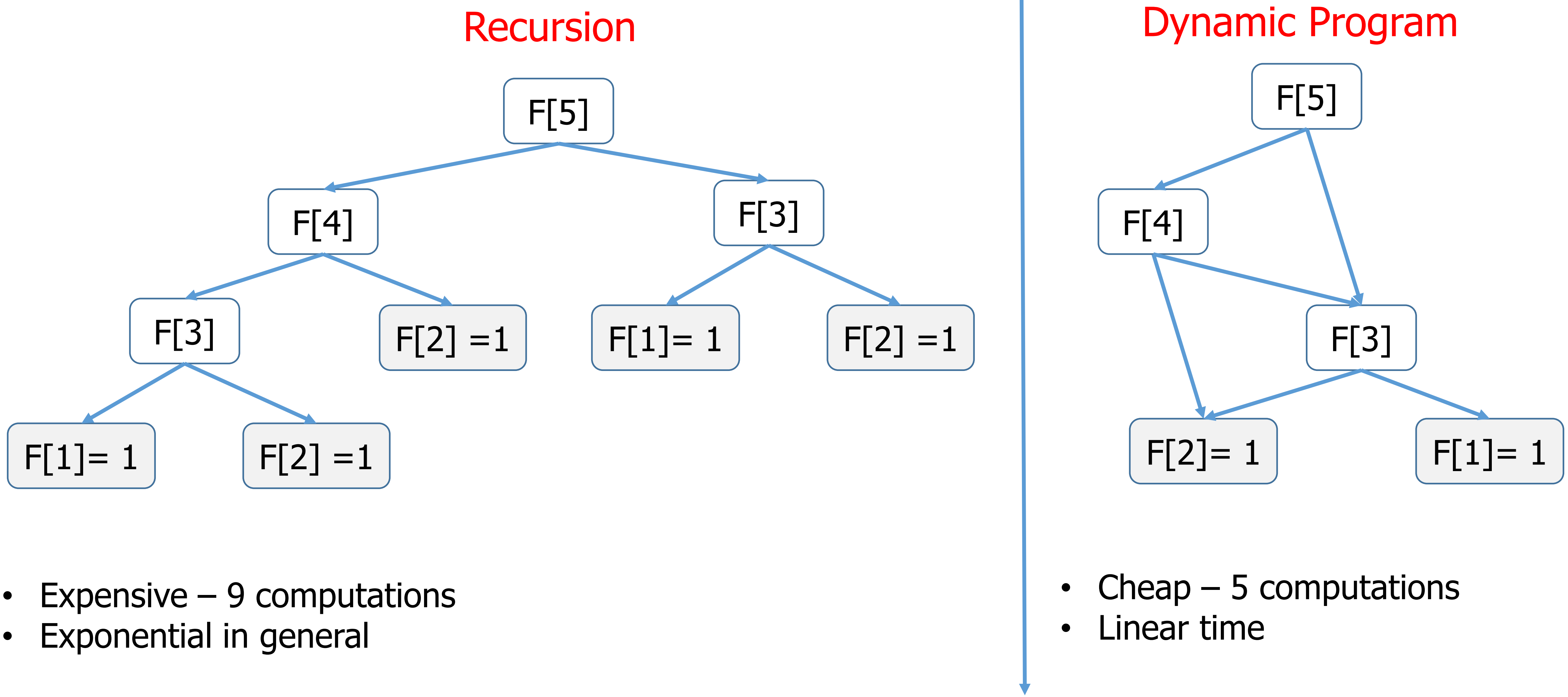
$$\boxed{F[n] = F[n-1] + F[n-2]}$$

F[1]	F[2]	F[3]	F[4]	F[5]	F[6]
1	1	2	3	5	8



Fibonacci Sequence – Dynamic Program

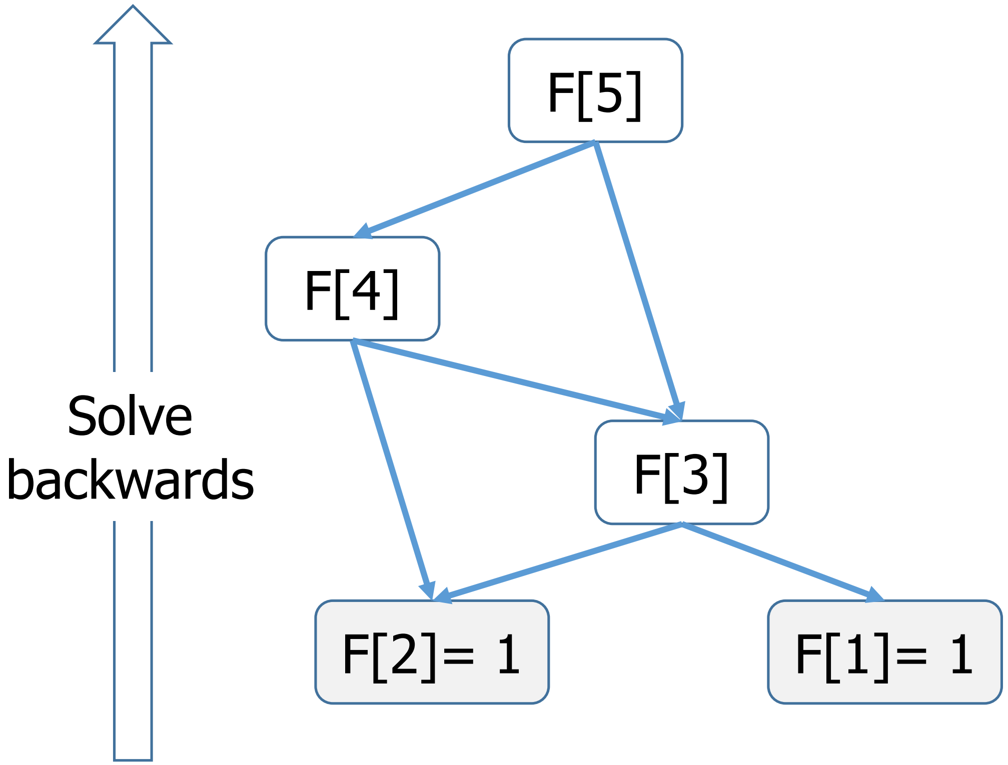
Problem: Given n, compute F[n]



- Expensive – 9 computations
- Exponential in general

- Cheap – 5 computations
- Linear time

Memoization




```
Input : n
Output : F[n]
    F[1] = 1
    F[2] = 1
    For j = 3 to n
        F [j] = F[j-1] + F[j-2]
```

Memoization : Latin for “remember”

Framework

- Identify sub-instances
- Identify recurrence relation
- Derive a bound on the number of sub-instances
  - Determines running time

- F[n] → F[n-1] and F[n-2]
- F[n] = F[n-1] + F[n-2]
- Number of sub-instances = n

- Sub-instances – may become non-trivial
- Recurrence relation – may become non-trivial
- DAG Structure – may become complex
- Correctness proof – may become non-trivial
- Deriving bound on number of sub-instances
  - May need combinatorial arguments and proofs
- May need to generalize the problem

- Why all this trouble?
  - Many seemingly exponential time problems admit fast, polynomial time dynamic programs

Subset Sum Problem

Subset Sum Problem

- Input : A sequence of numbers and a target T
- Objective : Does there exists a subset whose sum is exactly T? – no repetitions

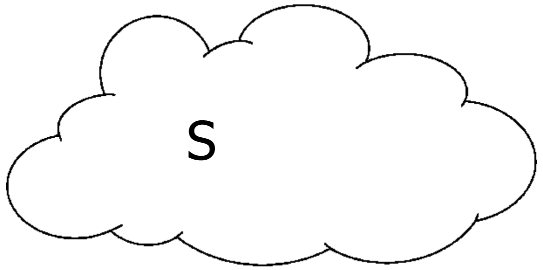
Input	5	3	2	6	8
T = 17	5	**3**	2	**6**	**8**
T = 13	**5**	3	**2**	**6**	8
T = 12	Not possible				

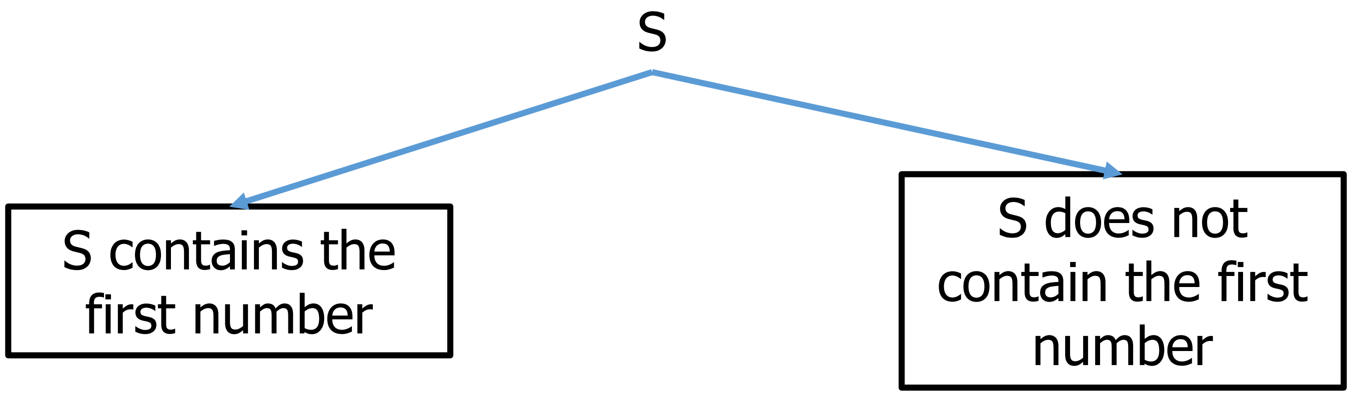
Naïve algorithm

- Try all possible subsets
- Check whether any of them yield T
- Running time: exponential – 2^n

Dynamic Program

- Input instance I
- What are the sub-instances we wish to create?
- Two Choices:
  - S does not contain the first number
  - S contains the first number

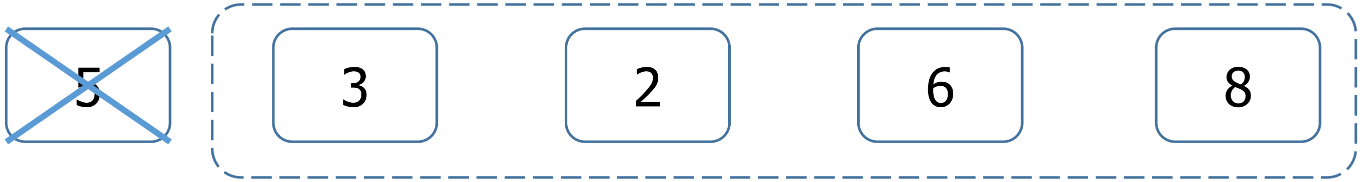




Two Choices

S does not contain the first number:

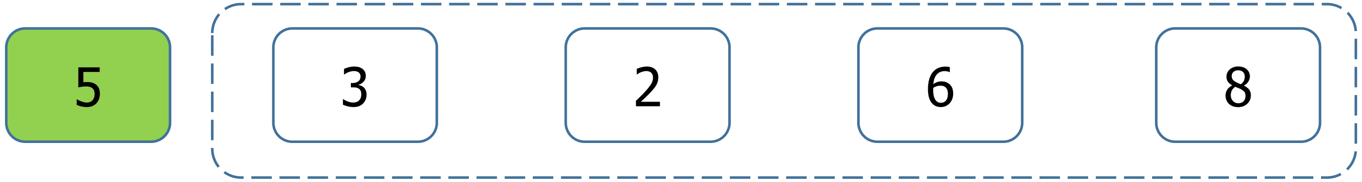
Sub-instance A1, Target = T



Lemma : I has solution if and only if A1 has solution

S contains the first number:

Sub-instance A2, Target = ?



Target = T - first number

Algorithm

Algorithm SSUM
Input : I = a1, a2, …. an and Target T
Output : Does there exist a solution of sum exactly T?

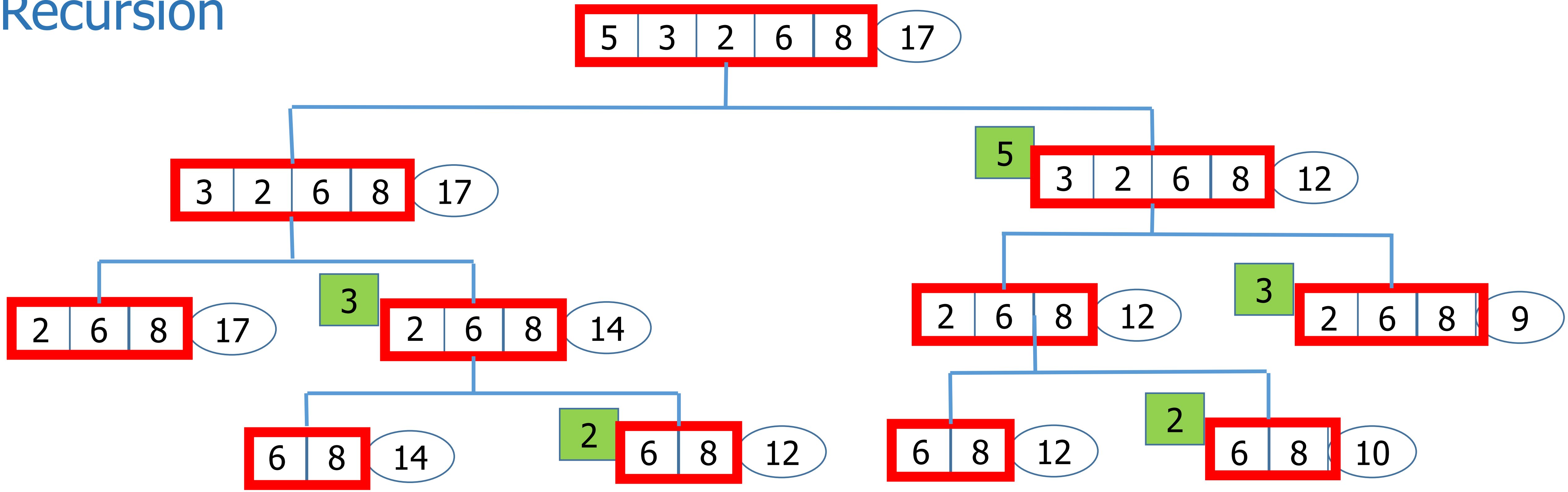
1. A1 = {a2, a3, ……, a_n} and target T
2. A2 = {a2, a3, ……., a_n} and target T -a1
3. Solve A1 and A2
4. Input has a solution if and only if A1 or A2 has a solution

$$\text{Solution}(I, t) = \left\{ \boxed{\begin{array}{c} \text{Solution}(I - a1, t) \\ \text{OR} \\ \text{Solution}(I, t - a1) \end{array}} \right.$$

Memoization:

- Whenever we solve a sub-instance, store its answer
- Whenever we want to solve a sub-instance, first check if we have solved it already

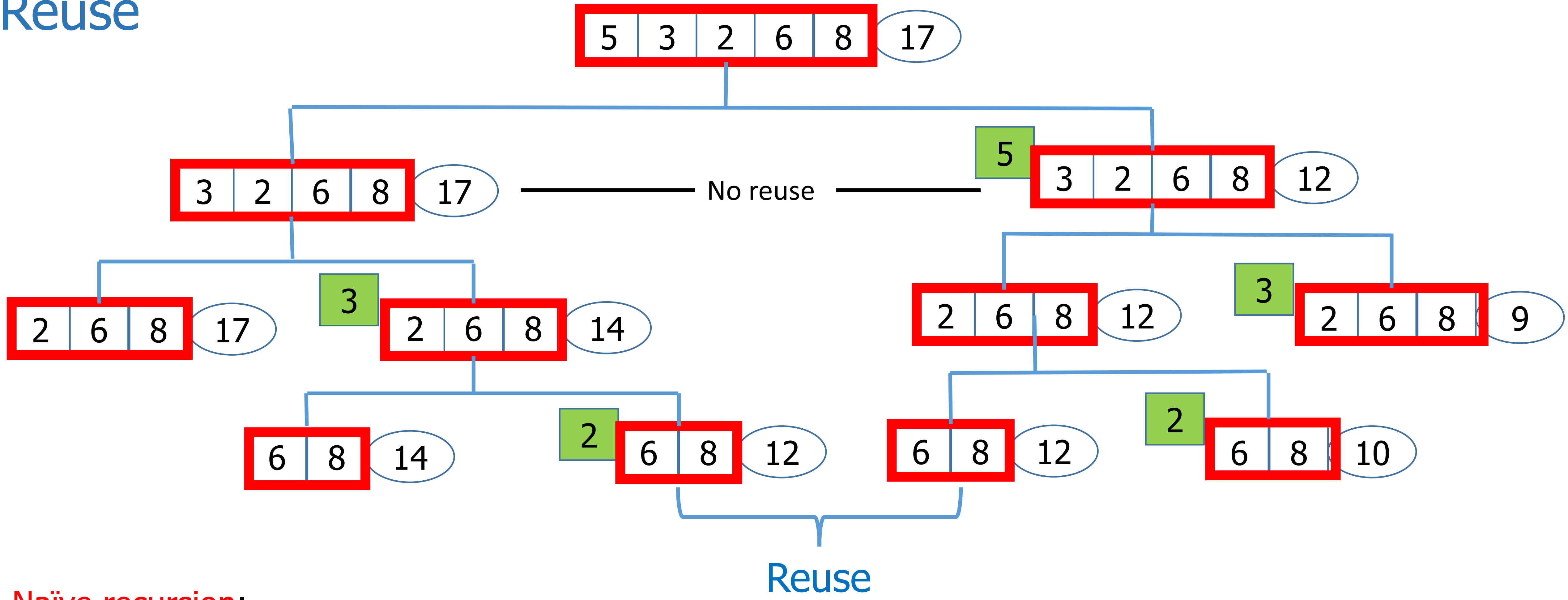
Recursion



Naïve recursion:

- Will grow exponentially
- 2^n nodes – one for each possible subset

Reuse



Naïve recursion:

- Will grow exponentially
- 2^n nodes – one for each possible subset

How many sub-instances?

That's all fine madam! But the question is:
How many sub-instances?

- Identify sub-instances
- Identify recurrence relation
- <u>Derive a bound on the number of sub-instances</u>
  - <u>Determines running time</u>

How many sub-instances?

- A sub-instances has [a subset, a target]

How many subsets?

- All subsets are suffixes of the input

At most n

How many targets?

- Any target is a number between 1 and original target T

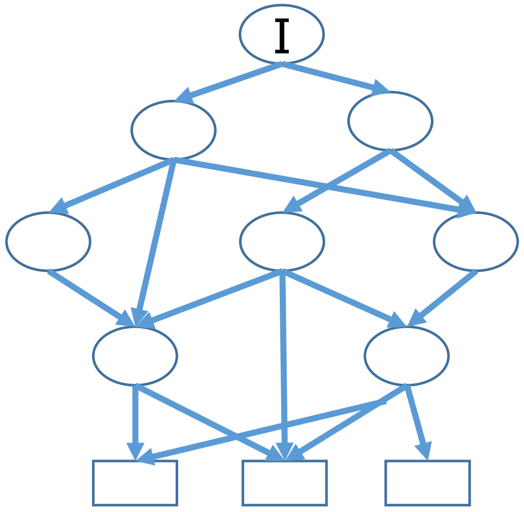
At most T

Lemma : Number of sub-instances is at most n x T

Theorem: Subset Sum problem can be solved in time O(nT)

Algorithm – with explicit memoization

- Each sub-instance can be represented as a pair < k, t >,
  - Represents suffix {a_k, ….., a_n}
  - Target = t
- Input : < 1, T >



Solve backwards

$$\text{Solution}(k, t) = \begin{cases} \text{Solution}(k+1, t) \\ \text{OR} \\ \text{Solution}(k+1, t - ak) \end{cases}$$

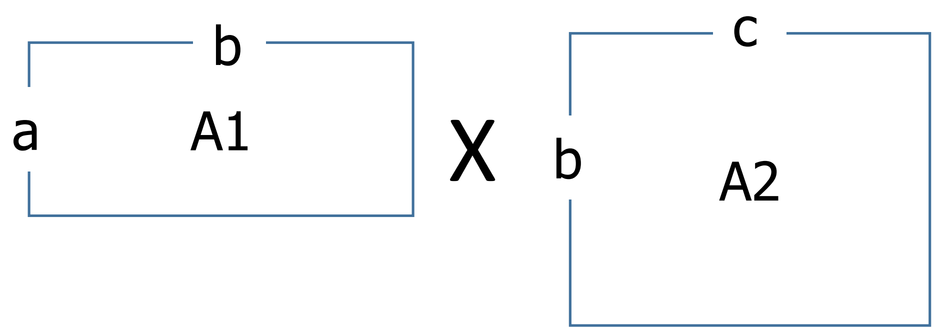
Algorithm SSUM
Input : a1, a2, …. an and Target T
Output : Does there exist a solution of sum exactly T?

```
For k = n to 1    Suffix starting point
    For t =  1 to T   target
        S(k, t) = S(k+1, t) OR S(k+1, t – ak)
```

Matrix Chain Multiplication

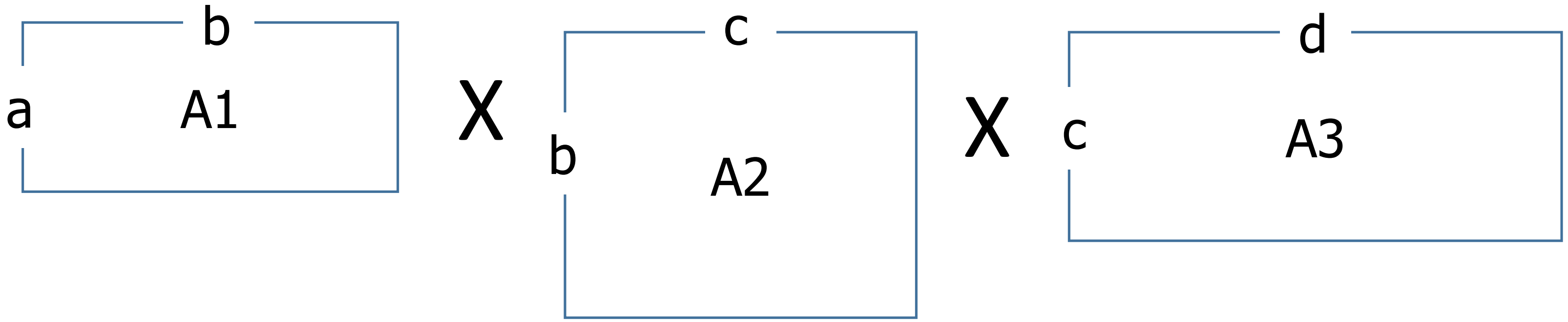
Matrix Chain Multiplication

- How many multiplications?

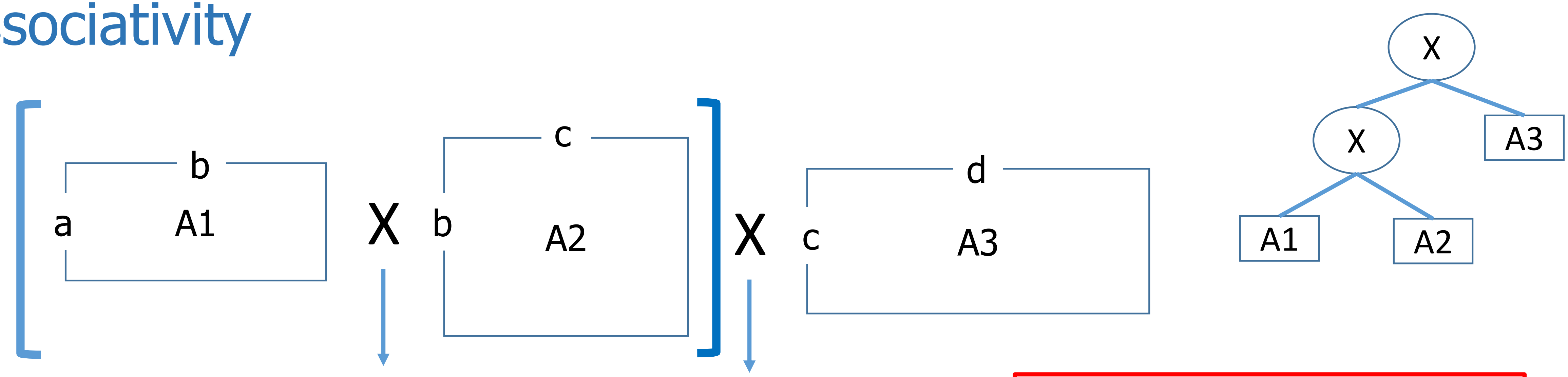


#multiplications = a x b x c

- How about three matrices?



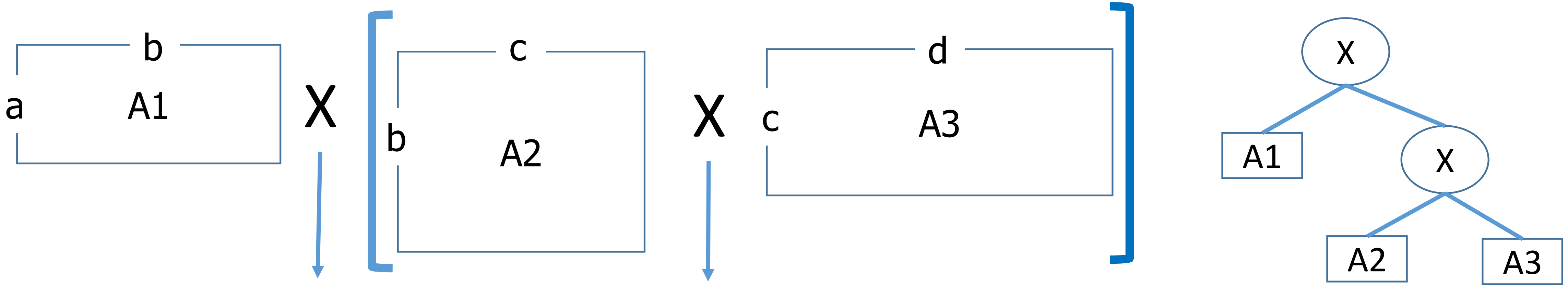
Associativity



#multiplications = abc
Output = a x c matrix

#multiplications = acd
Output = a x d matrix

#multiplications = abc + acd

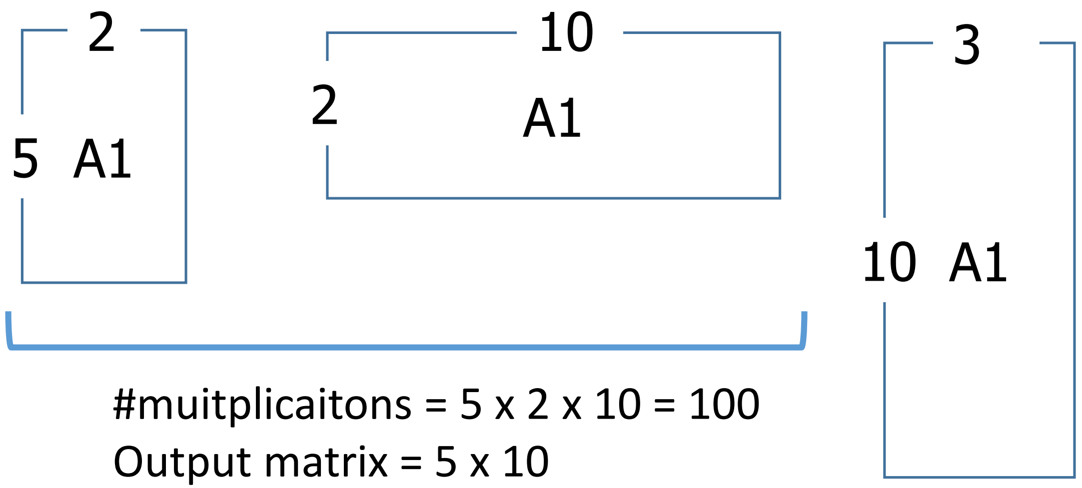


#multiplications = abd
Output = a x d matrix

#multiplications = bcd
Output = b x d matrix

#multiplications = abd + bcd

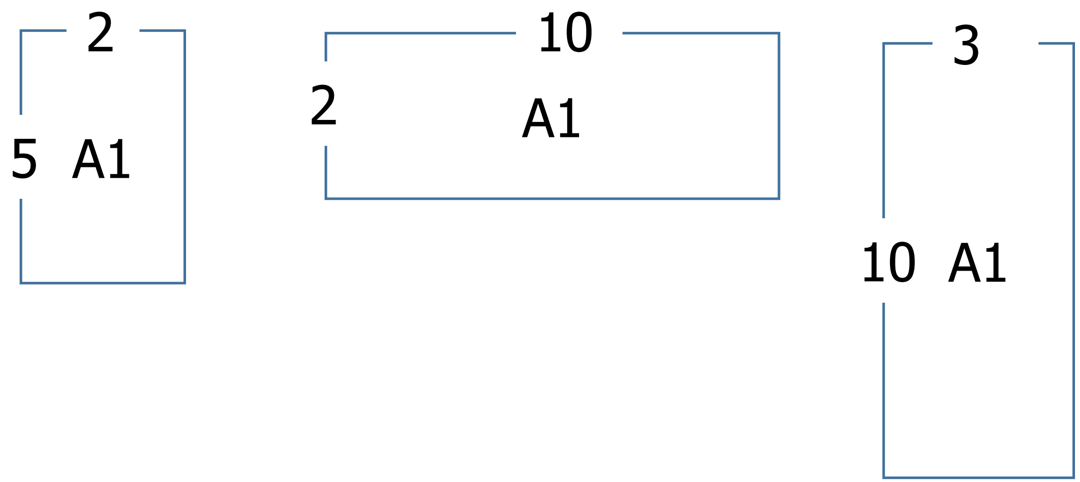
Example



#muitplicaitons = 5 x 2 x 10 = 100
Output matrix = 5 x 10

#muitplicaitons = 5 x 10 x 3= 150
Output matrix = 5 x 3

Total multiplications = 100 + 150 = 250



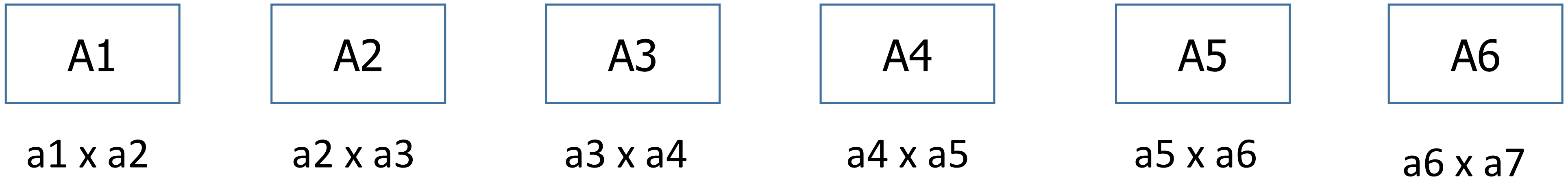
#muitplicaitons = 2 x 10 x 3= 60
Output matrix = 2 x 3

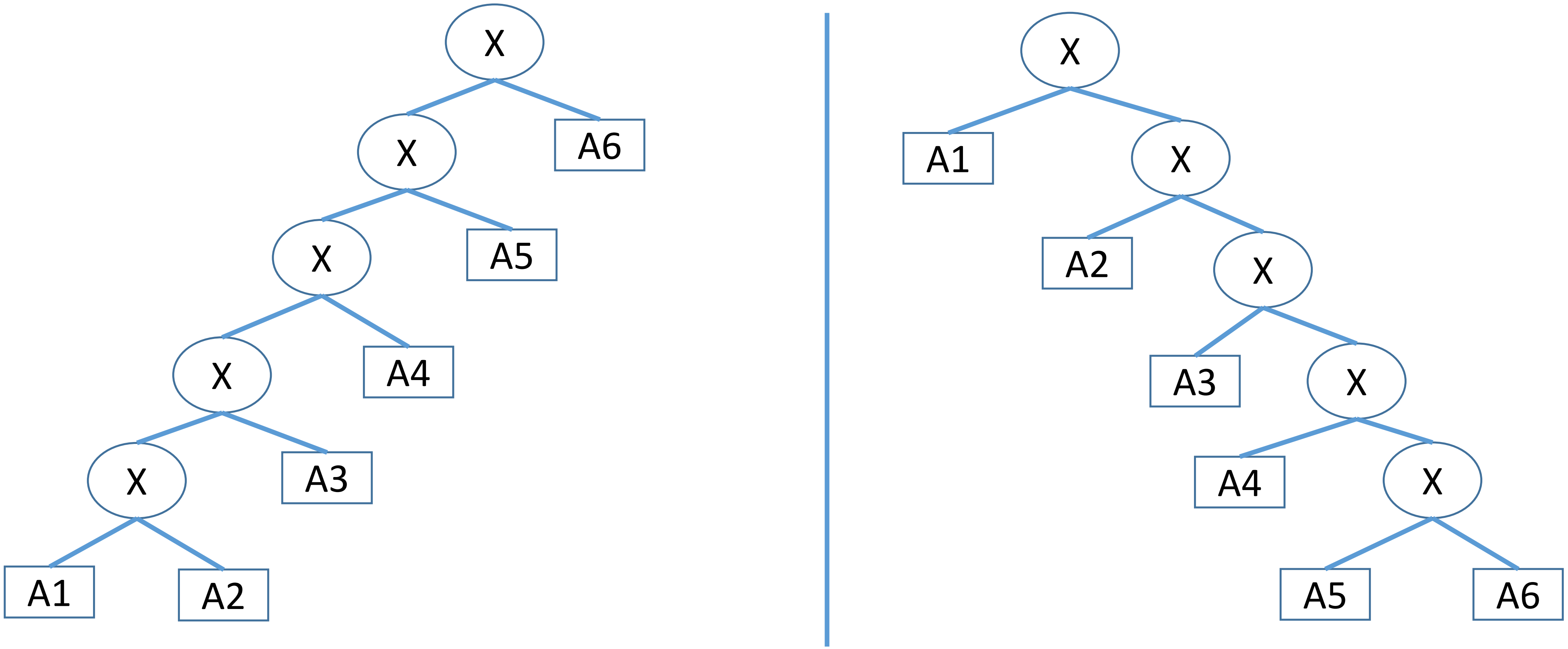
#muitplicaitons = 5 x 2 x 3= 30
Output matrix = 5 x 3

Total multiplications = 60 + 30 = 90

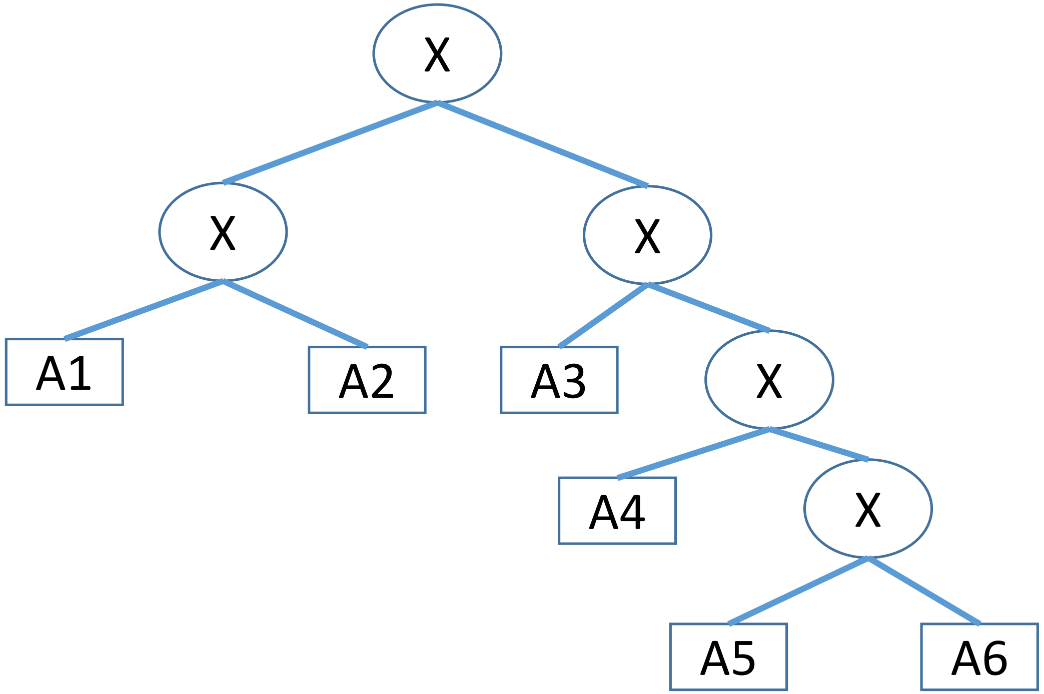
Matrix Chain Multiplication

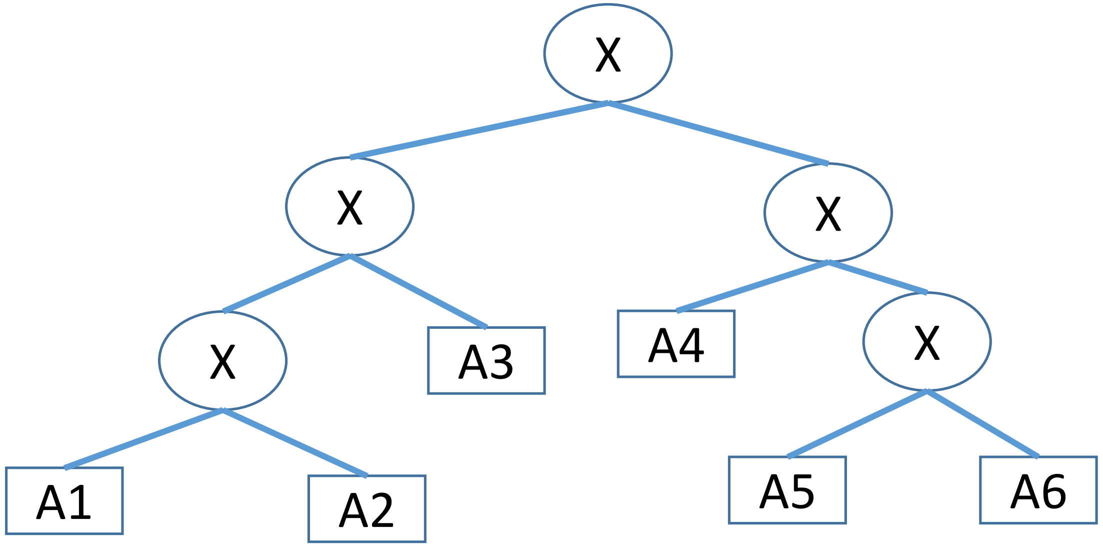
Input: A sequence of matrices
Output: Plan for multiplication so that total number of multiplications is minimized





More choices



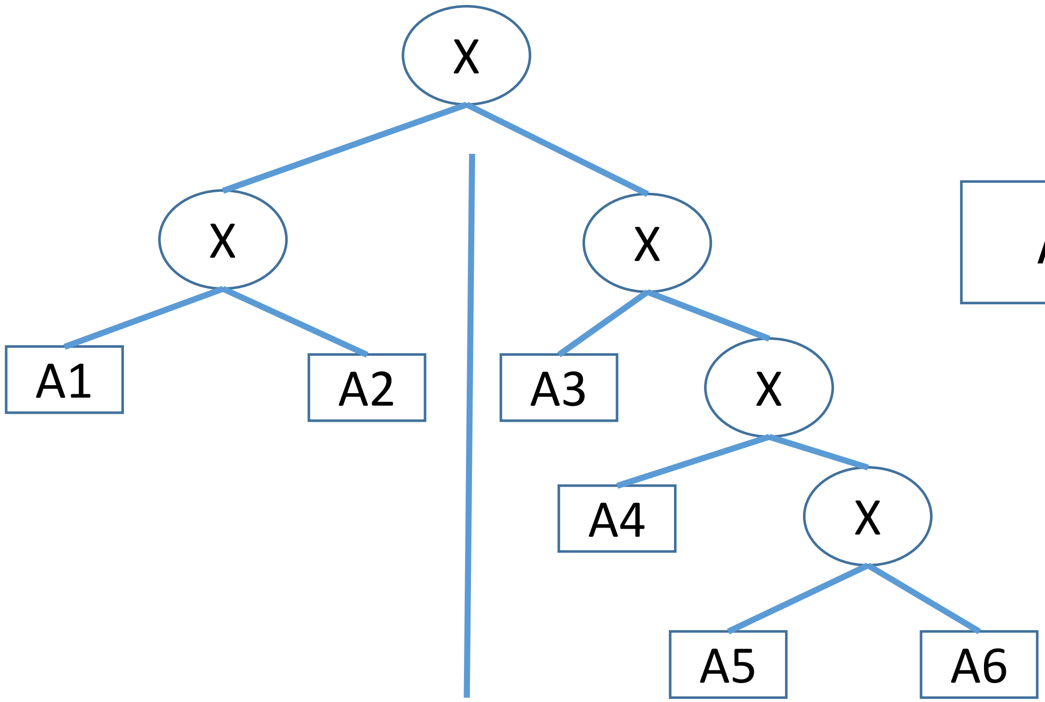


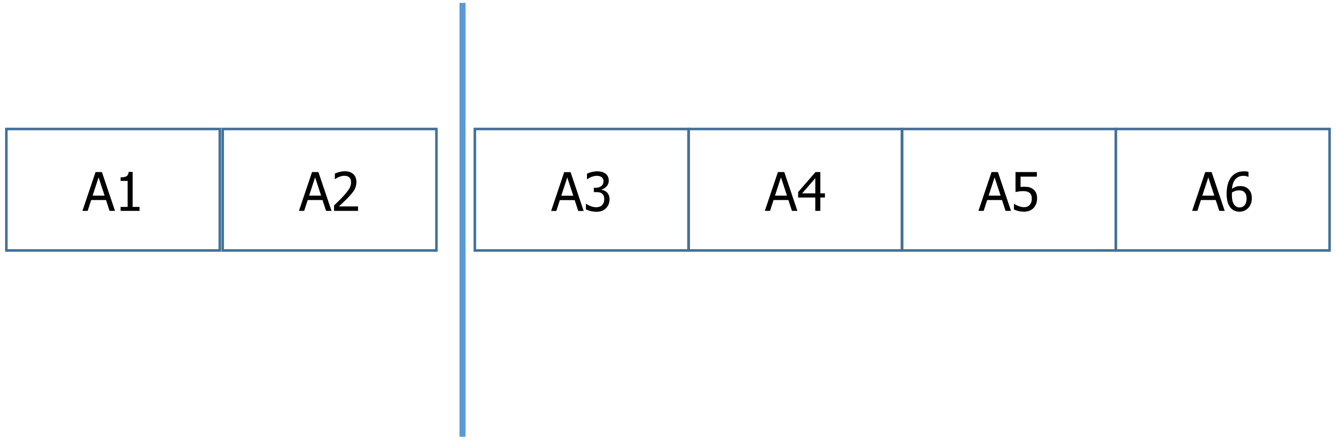
How many choices?

- Catalan number: $\frac{1}{n+1}\binom{2n}{n} \approx 2^n$

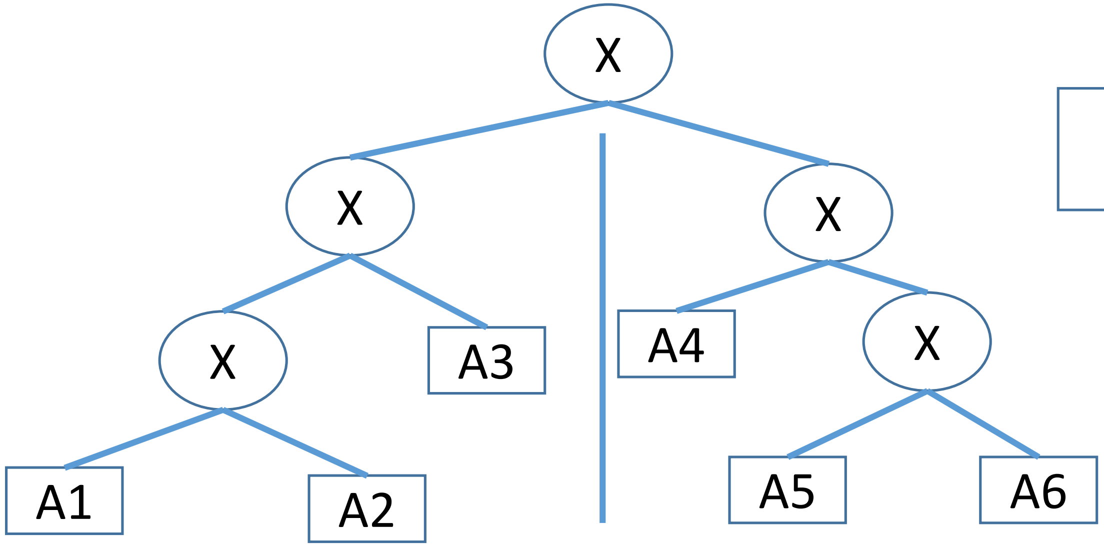
Eugene Catalan

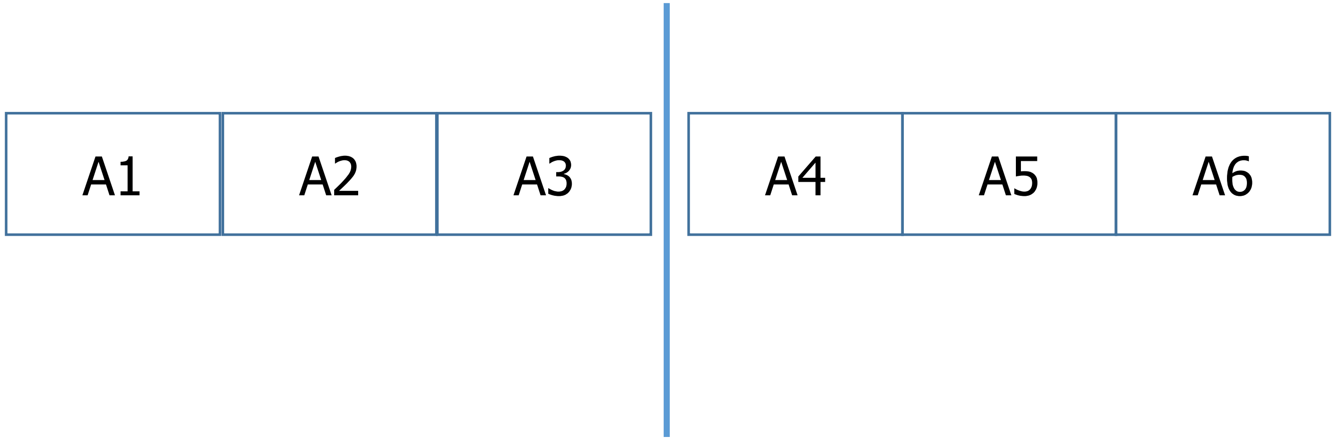
Dynamic Programming





We are going to guess this top-most split





Algorithm

Algorithm: MCM
Input: A1, A2, ….., An
Output: Plan

1. t = GET_BEST_SPLIT
2. X = MCM(A1, …, At)
3. Y = MCM(A_t+1, …, An)
4. X * Y

t

A1 A2 A3 A4 A5 A6

a1 x a2 a2 x a3 a3 x a4 a4 x a5 a5 x a6 a6 x a7

X

#muitplicaitons = Cost(X)
Output matrix = a1 x a4

Y

#muitplicaitons = Cost(Y)
Output matrix = a4 x a7

Cost(X * Y) = a1 x a4 x a7
Cost(X * Y) = a1 x a_t+1 x a_n

How to get the best split?

Try all possibilities : there are only n of them

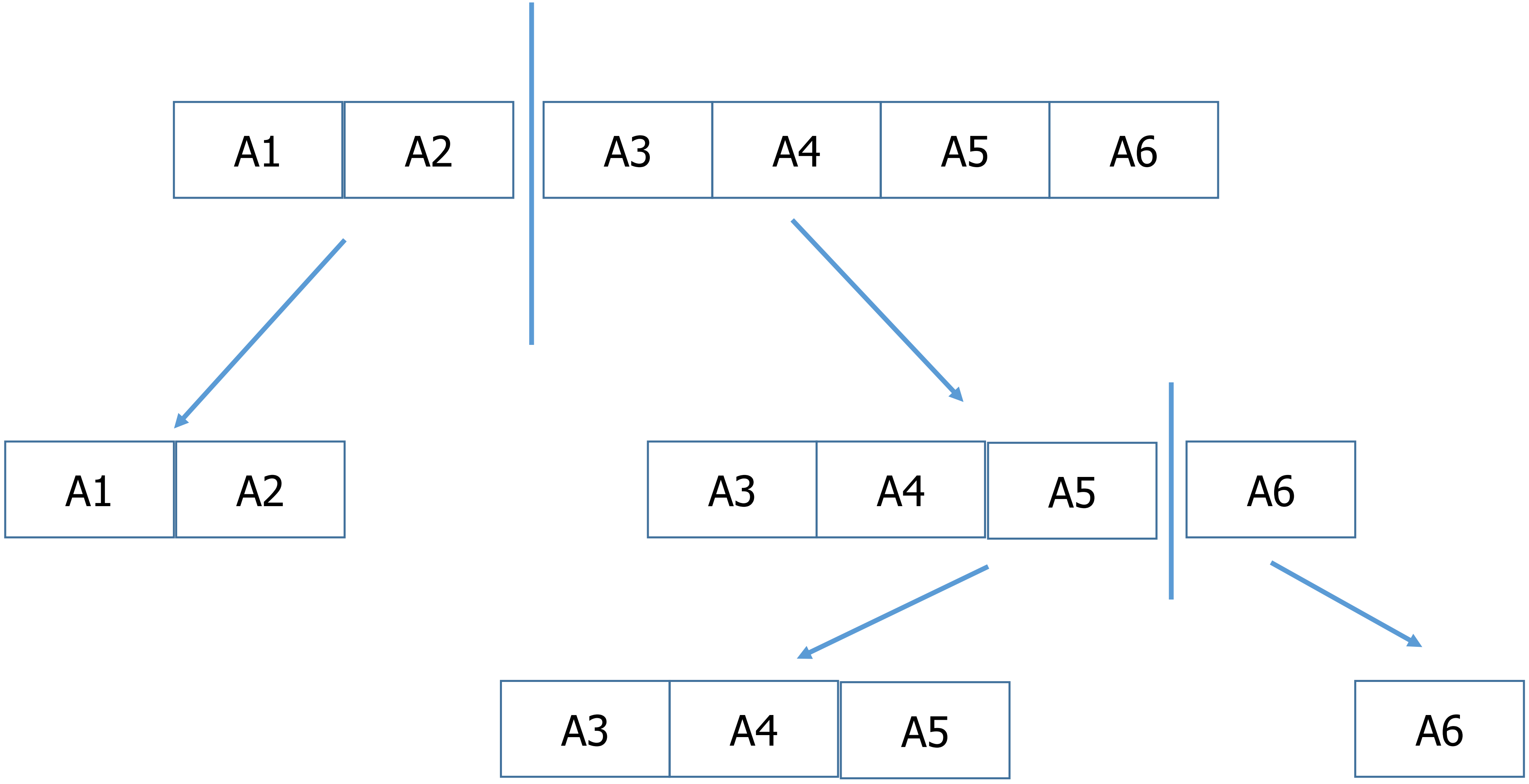
Algorithm: MCM
Input: A1, A2, ….., An
Output: Plan

For each t = 1 to n-1
    X = MCM(A1, …, At)
    Y = MCM(A_t+1, …, An)
    X * Y
    Cost = Cost(X) + Cost(Y) + Cost(X * Y)
Among the t choices choose the best

$$Cost(A_1 \ \ldots., A_n) = \min_{t} \left[\begin{array}{c} Cost(A_1, \ldots. A_t) \\ + \\ Cost(A_{t+1}, \ldots. A_n) \\ + \\ a_1 a_{t+1} a_n \end{array} \right]$$

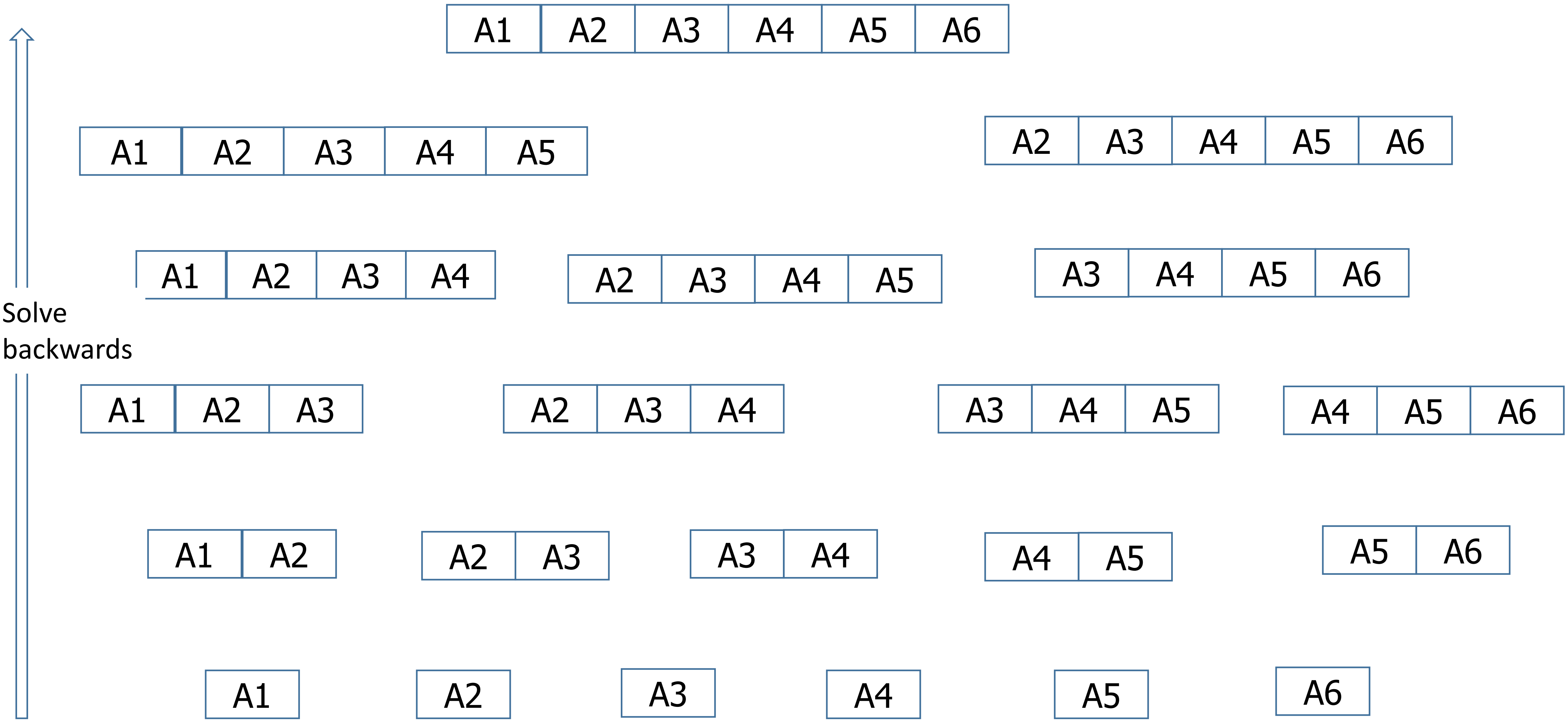
- More complex aggregation : min over sums

Sub-problem structure



Problem instance
Not a prefix or a suffix.
A middle segment!

Sub-problem structure



Algorithm

```
For L = 1 to n                         Length of segment
    For s = 1 to n – L                 Start of segment
        e = s + L - 1                  End of segment
        Cost[s, s+L] = infinity        Initialize
        For t = s to e-1               Try all possible splits
            C = Cost[s, t] + Cost[t+1, e] + a_s x a_t+1 x a_e     Cost of this split
            If C is smaller than Cost[s, s+L],     Is this a better split?
                Cost[s, s+L] = C       If so, take it
```

- Number of entries = number of segments = n^2
- Number of splits per entry = n
- Total running time = O(n^3)

Theorem: Our algorithm finds the optimal solution. Its running time is at most O(n^3)

Largest monotone subsequence

Largest Monotone Subsequence

- Input : A sequence of numbers
- Sub-sequence : a selection of the numbers
- Monotone : they are in increasing order
- Objective : Find the largest – having the largest sum

Input	10	8	25	9	21	14	20	7

Monotone : value = 44

10	8	25	9	21	**14**	**20**	7

Not monotone

10	**8**	25	9	**21**	**14**	20	7

Monotone : value = 51

10	**8**	25	**9**	21	**14**	**20**	7

Play with it a bit!

Attempt 1: Start from first number and keep going higher

10	8	25	9	21	14	20	7

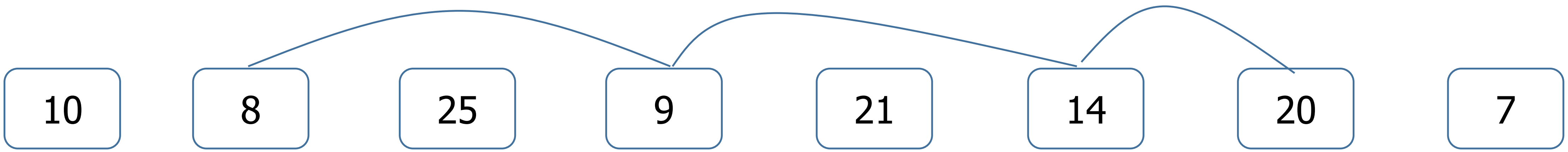
Attempt 2: Try starting from each number and keep going higher!

10	8	25	9	21	14	20	7

10	8	25	9	21	14	20	7

Optimal Solution

Optimal Solution: value = 51



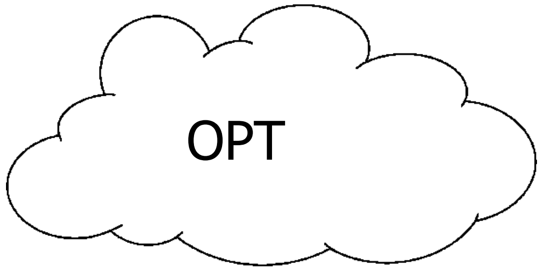
- Optimal solution - may not be greedy. May skip numbers.
- How to determine the skips?

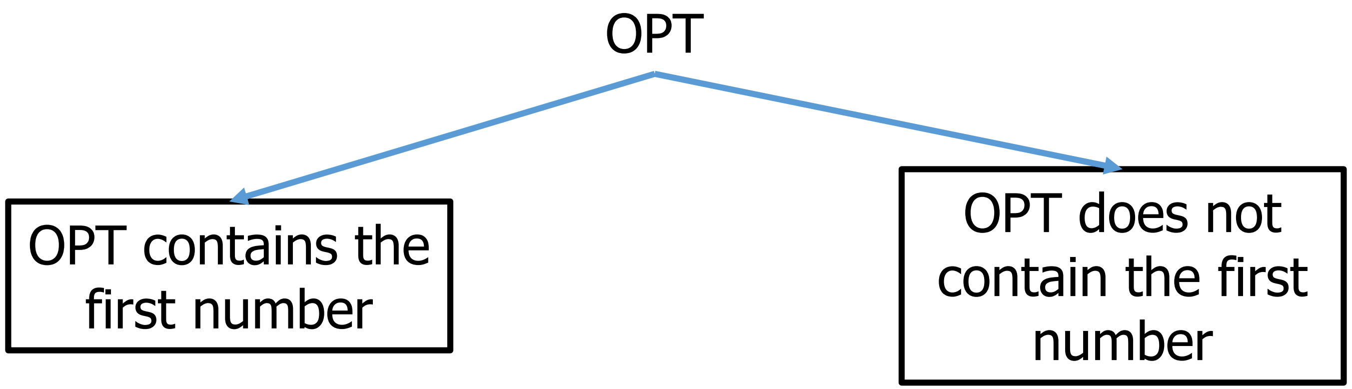
Naïve algorithm

- Try all possible subsequences
- Filter out non-monotone subsequences
- Among the monotone subsequences – choose the best
- Running time: exponential – 2^n

Dynamic Program

- Input instance I
- What are the sub-instances we wish to create?
- Two Choices:
  - OPT does not contain the first number
  - OPT contains the first number





Two Choices

OPT does not contain the first number:

Sub-instance A1

~~10~~ | 8 | 25 | 9 | 21 | 14 | 20 | 7

Lemma : Any solution to A1 is also a solution to input instance I

- Find optimal solution to A1.
- Output it.

OPT(I) = OPT(A1)

Two Choices

OPT contains the first number:

10 ~~8~~ 25 ~~9~~ 21 14 20 7

Sub-instance A2

25 21 14 20 ~~7~~

Lemma : Given any solution to A2, we can obtain a solution to I by pre-pending the first number

10 | 25 21 14 20

- Find optimal solution to A2.
- Prepend the first number.

OPT(I) = First number + OPT(A2)

Algorithm

Algorithm MonSeq
Input : a1, a2, …. an
Output : Largest monotone Sequence

1. A1 = {a2, a3, ……, a_n}
2. A2 = {a_i : a_i >= a_1, i >= 2}
3. Optimal solution Opt1= **MonSeq**(A1)
4. Optimal solution Opt2= **MonSeq**(A2)
5. S1 = Opt1
6. S2 = a1 + Opt2
7. Output best of S1 and S2

Opt(I) = Max(
Opt(a_2, a_3, …., a_n),
a_1 + Opt(a_i : a_i >= a_1, i >= 2)
)

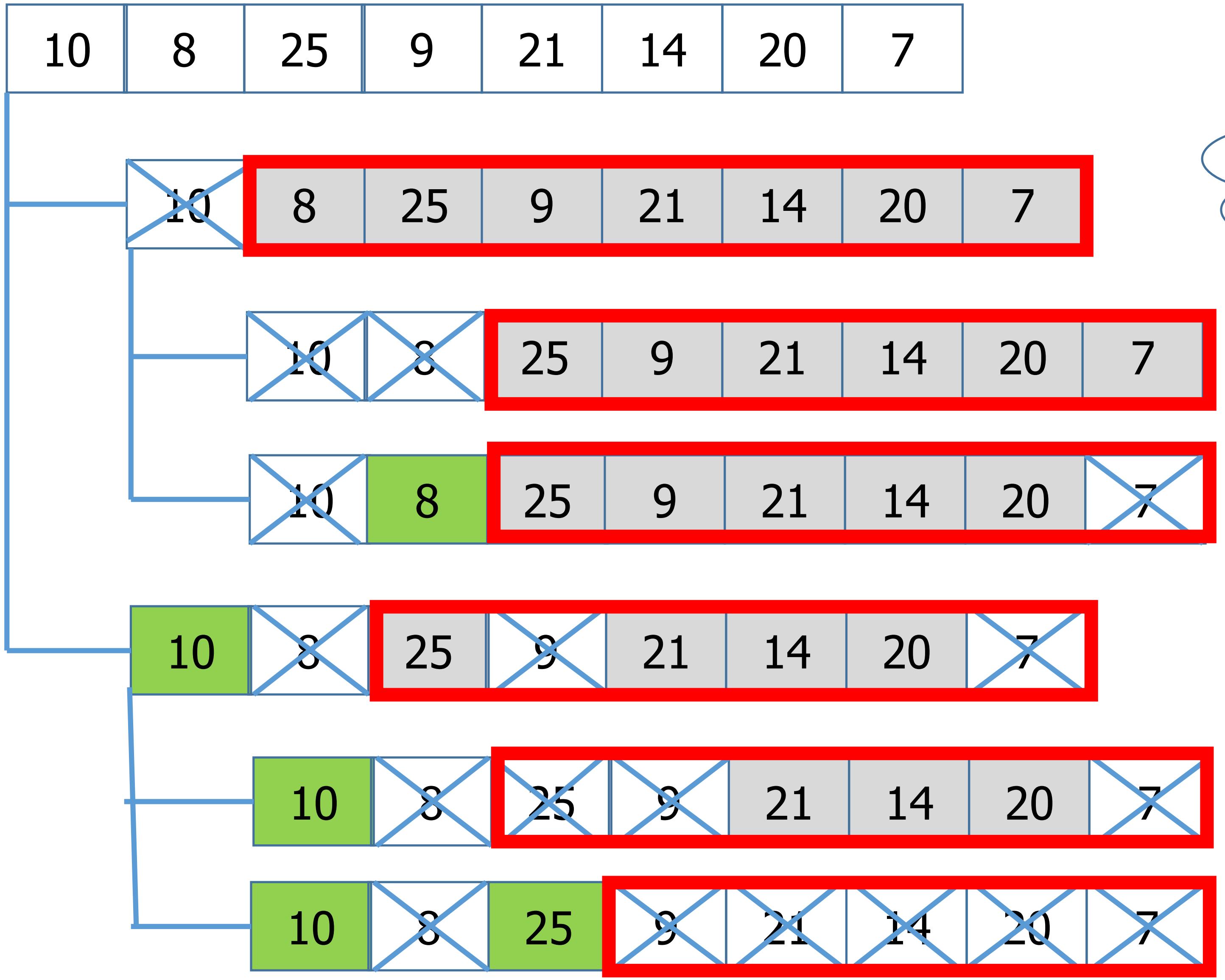
Memoization:
- Whenever we solve a sub-instance store its optimal solution
- Whenever we want to solve a sub-instance, first check if we have solved it already

How many sub-instances?

That’s all fine madam! But the question is:
How many sub-instances?

- Identify sub-instances
- Identify recurrence relation
- <u>Derive a bound on the number of sub-instances</u>
    - <u>Determines running time</u>

How many sub-instances?



Are we getting all kinds of subsets as sub-instance?

- If that is the case, we are doomed
  - 2^n subsets
  - Exponential time

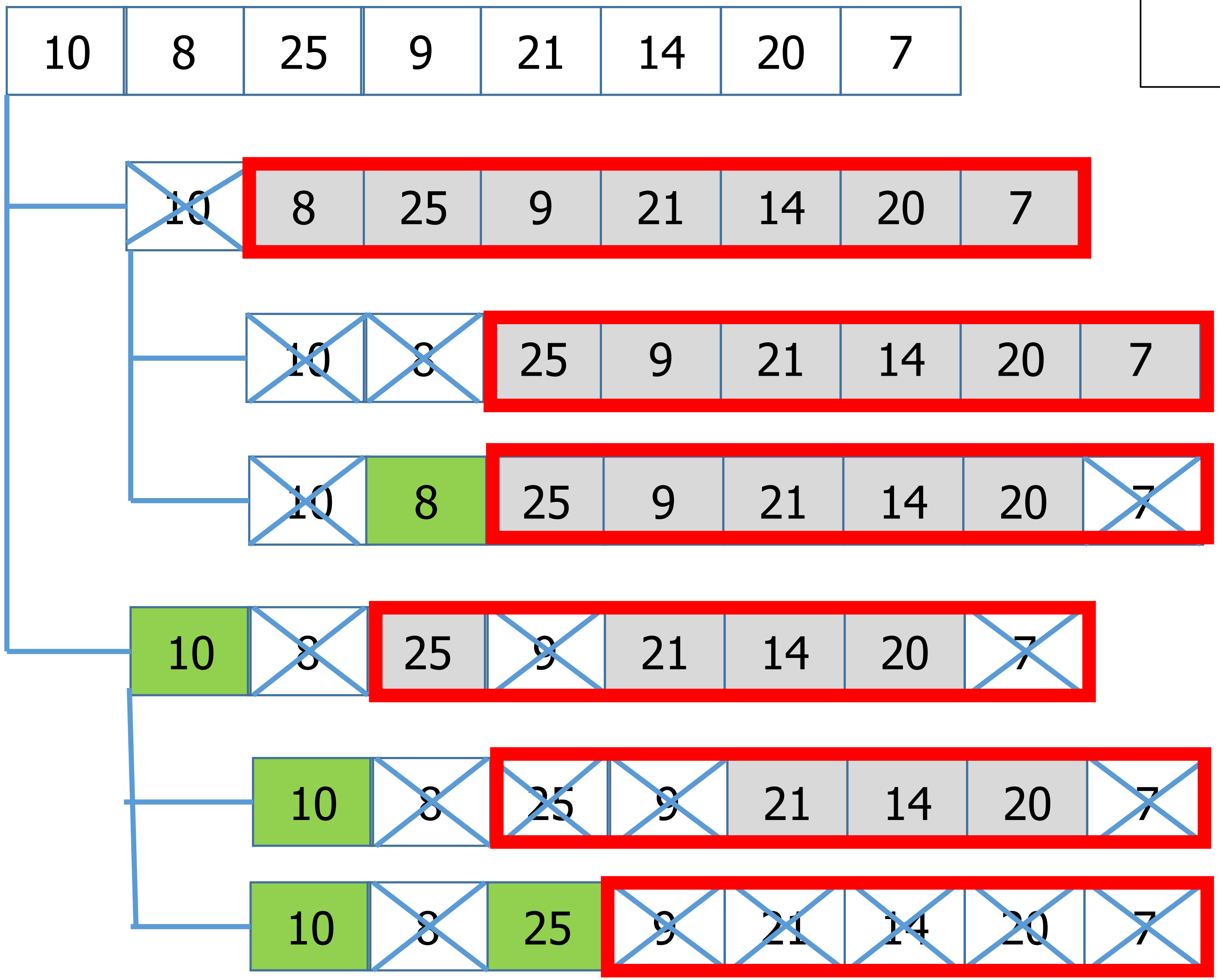
Two methods to proceed

- Derive a bound on the number of sub-instances
- Generalize the problem

Hay! It is not hard.
Don’t make a fuzz out of it!

How many sub-instances?

Each sub-instance
- Is a suffix of the original sequence
- But some numbers may go missing



No number is missing

No number is missing

Smaller than 8 are missing

Smaller than 10 are missing

Smaller than 10 are missing

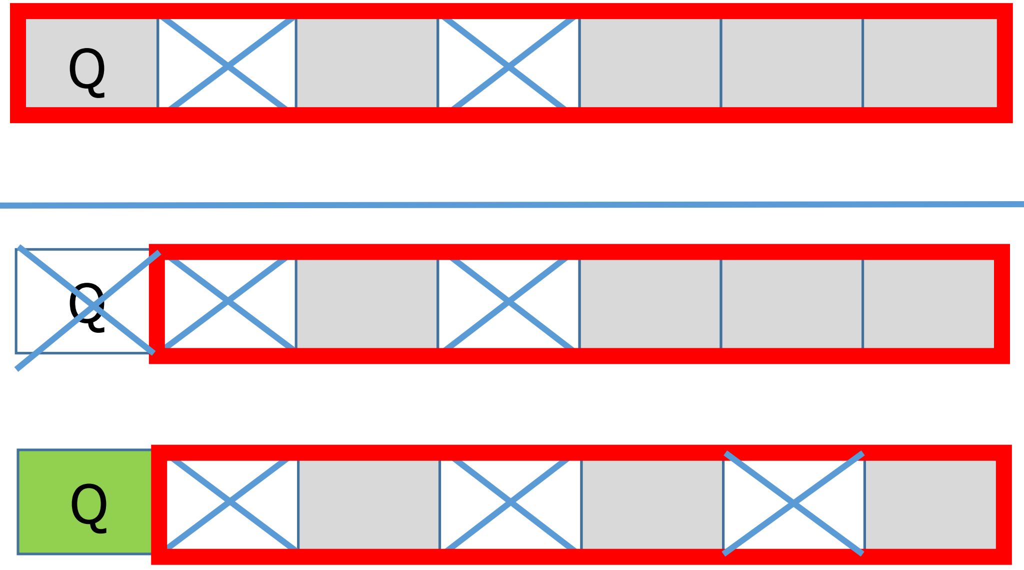
Smaller than 25 are missing

Sub-instance Structure

Lemma : Each sub-instance is a
- Suffix of the original sequence with some numbers missing
- All missing numbers are smaller than a prior number → call it the *pivot*

Proof

- Suppose the property is true up to some stage
- Let us see what happens in the next stage



Pivot P
- All missing numbers are less than P
- Q is larger than P

Pivot P
- P continues to be the pivot

Pivot Q
- Q becomes the pivot

How many sub-instances?

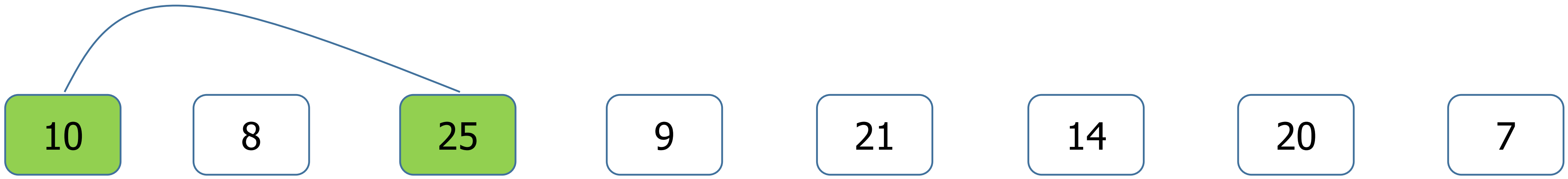
Lemma : Each sub-instance is a
- Suffix of the original sequence with some numbers missing
- All missing numbers are smaller than a prior number → call it the *pivot*

- Number of suffixes possible? At most n
- Number of pivots possible? At most n
- Number of sub-instances possible? At most n^2

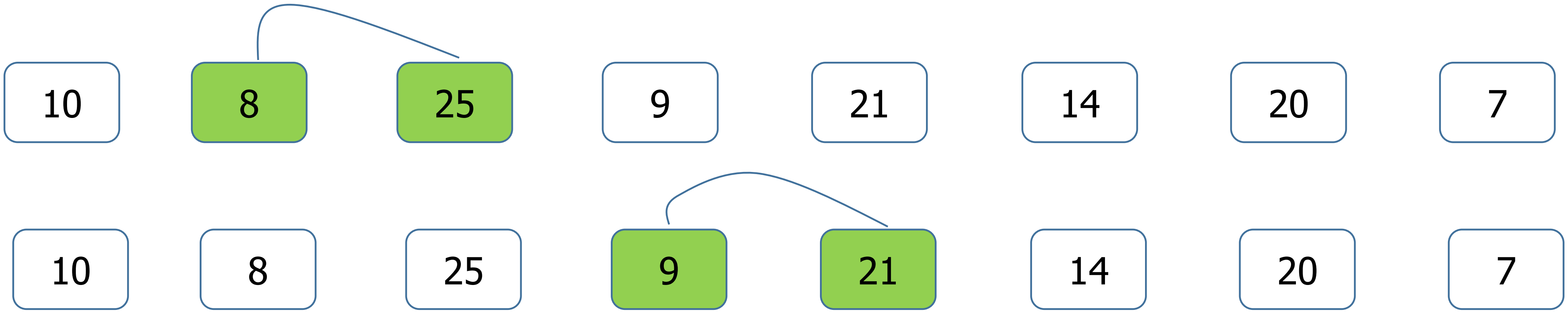
Theorem: Our algorithm finds the optimal solution. Its running time is at most O(n^2)

Comparison to Greedy Methods

Attempt 1: Start from first number and keep going higher



Attempt 2: Try starting from each number and keep going higher!



Comparison to Greedy Method – Dynamic Programming

~~10~~ 8 25 9 21 14 20 7

Do not greedily pick 25 – You may miss better choices later

10 8 25 9 21 14 20 7

Instead look for optimal solution for suffix starting at 25

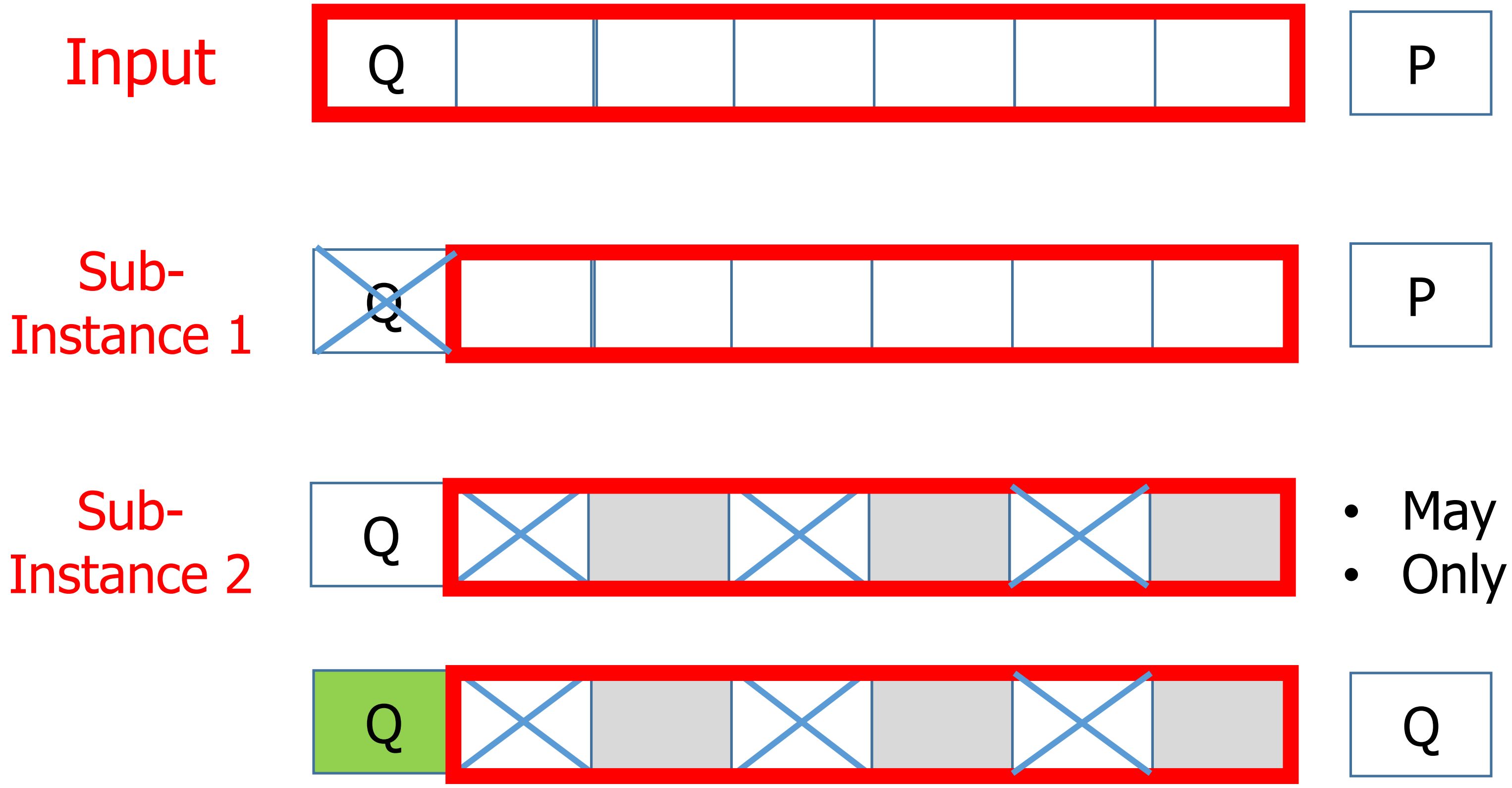
10 8 25 9 21 14 20 7

- Optimal solution for suffix starting at 25 – May or may not pick.
- Leave it to recursion to decide

Method 2: Generalize the problem

Pivoted Largest Monotone Subsequence Problem

- Input : A sequence of numbers and a number P called the pivot
- Objective :
    - Find the largest monotone sub-sequence involving only numbers larger than P
        - You are allowed to use only the numbers larger than P



- May not even be possible
- Only possible if Q > P.

Algorithm for Pivoted Largest Monotone Subsequence

$$
\text{OPT}(I, P) = \begin{cases} \text{OPT}(I\text{-}Q, P) & \text{If } Q < P \\ \text{Max} \begin{bmatrix} \text{OPT}(I\text{-}Q, P) \\ Q + \text{OPT}(I\text{-}Q, Q) \end{bmatrix} & \text{If } Q > P \end{cases}
$$

- Number of suffixes possible?

  At most n

- Number of pivots possible?

  At most n

- Number of sub-instances possible?

  At most n^2

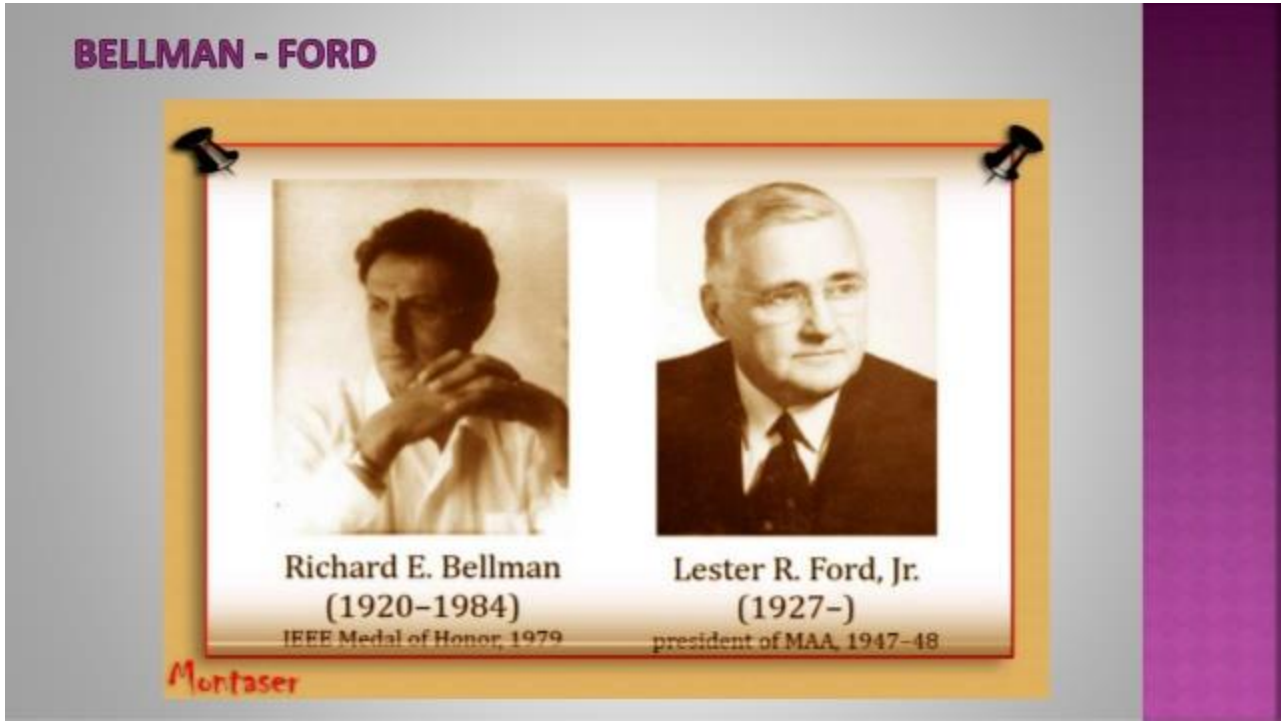
Algorithm for Largest Monotone Subsequence

Input: Sequence I

Solution:

- Solve Pivoted Largest Monotone Subsequence Problem
- With input I and Pivot = infinity

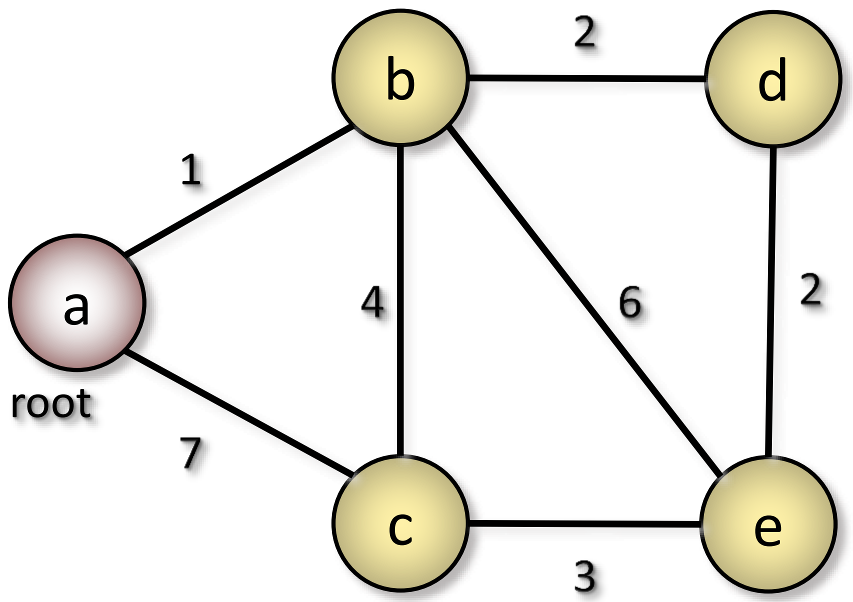
Single Source Shortest Path Problem
Bellman Ford Algorithm

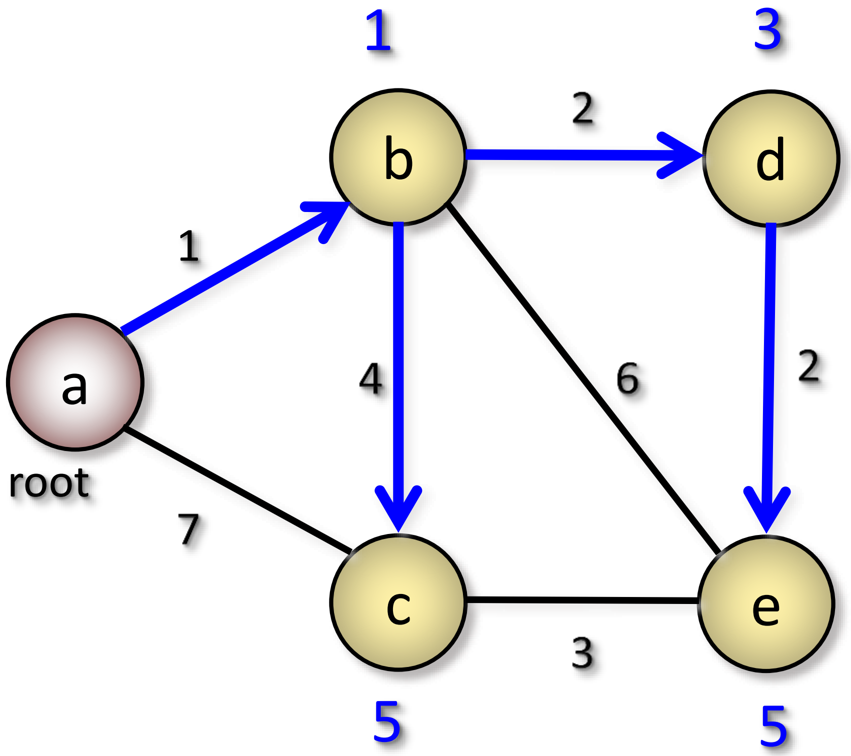


Single Source Shortest Path: Bellman-Ford Algorithm

Input:
- An undirected graph
- Weights on the edges
- A root vertex

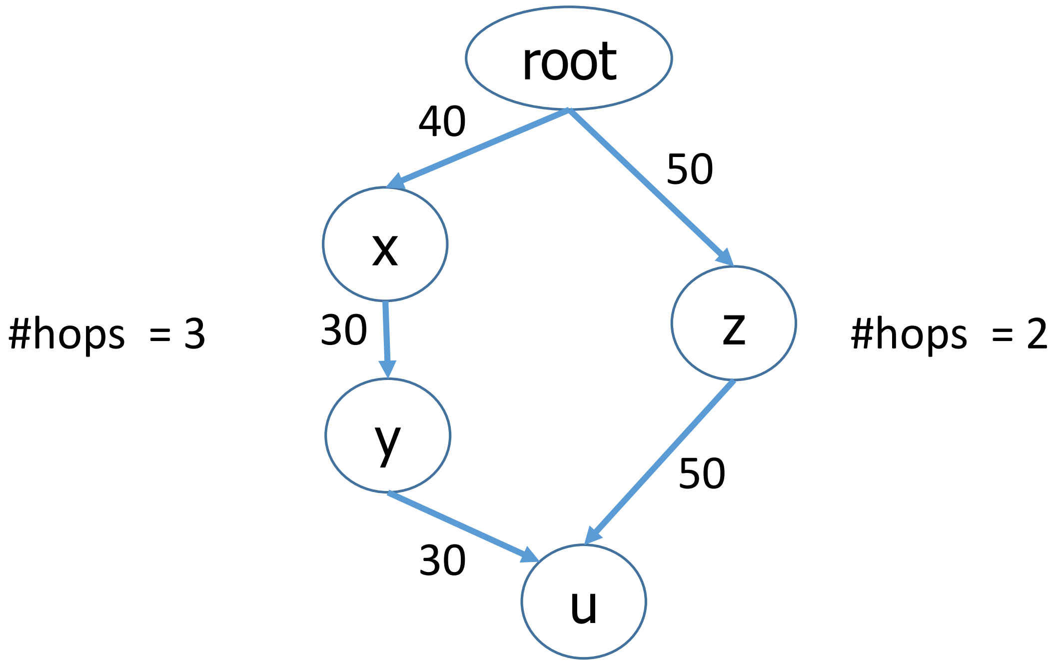
Output:
- Shortest path from the root to all the other vertices





50

Distance vs Hops

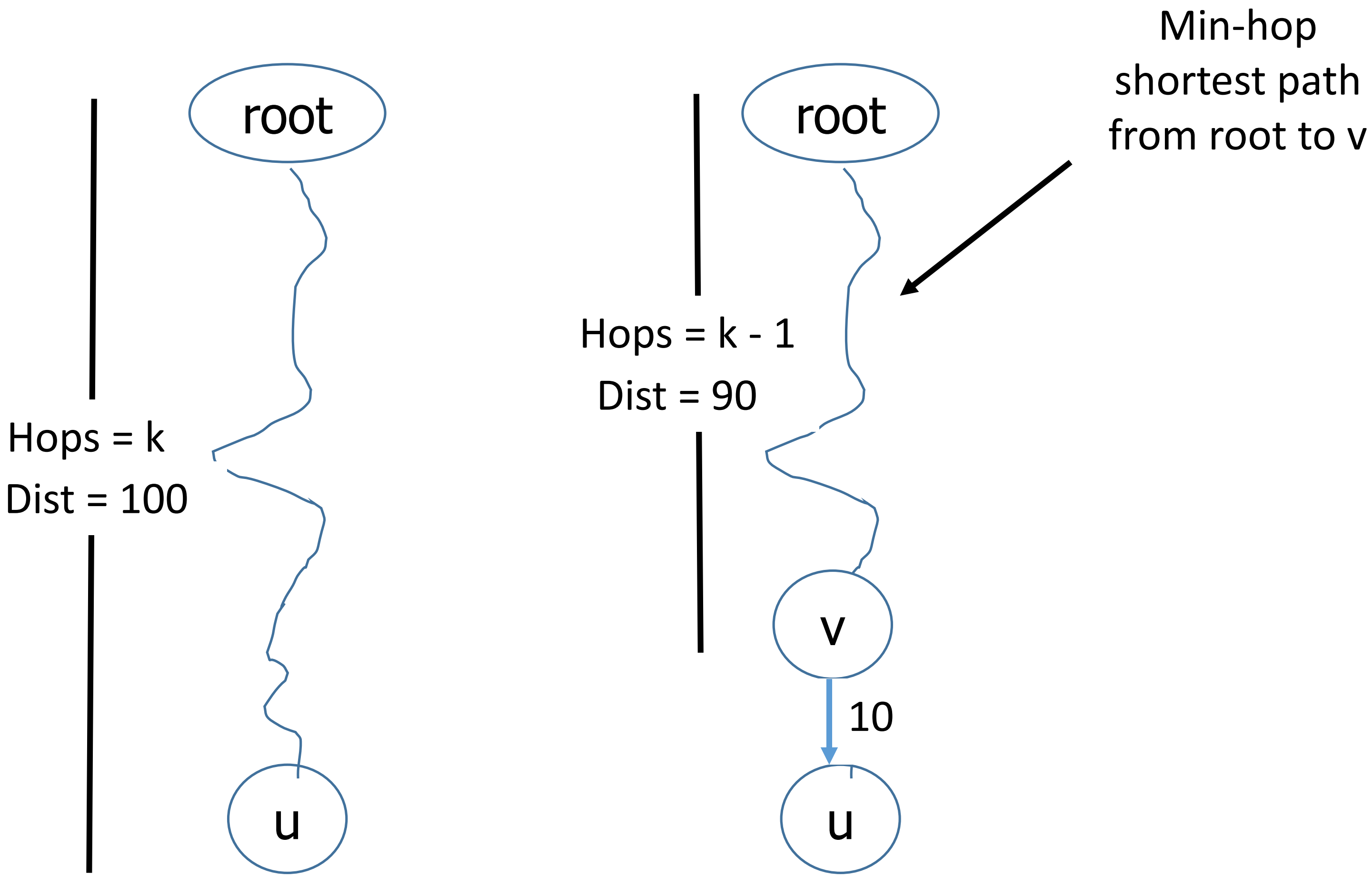


- Shortest path may not have the least hops
- There may be many shortest paths with different hop count

MinHop(u) = minimum number hops among the shortest paths

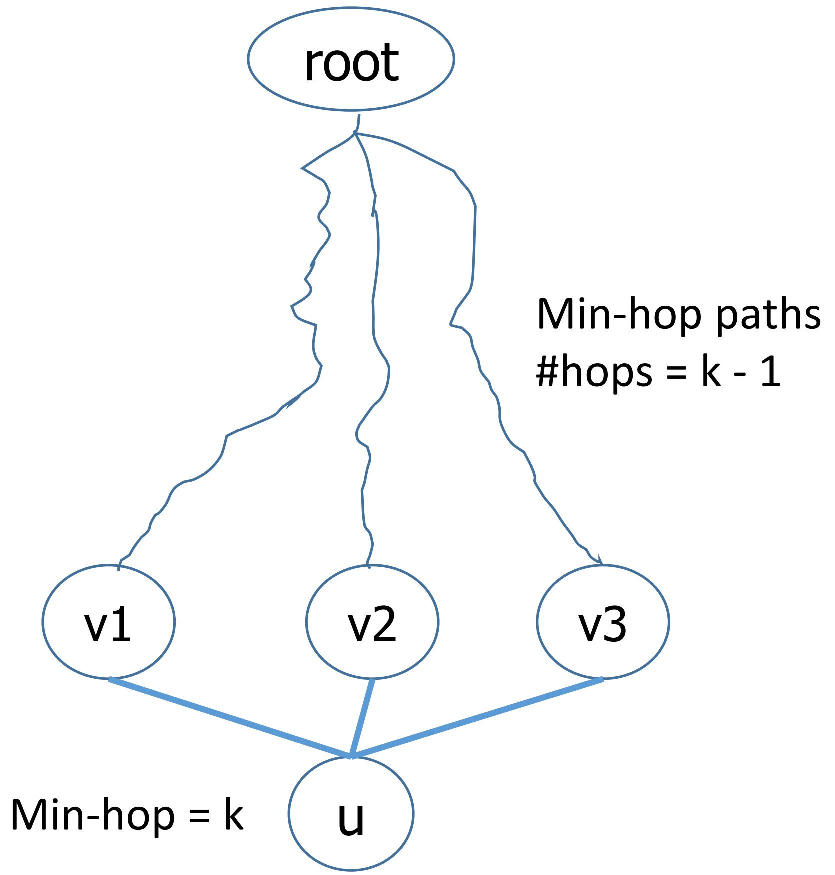
Main Observation

- Min-hop shortest path to u



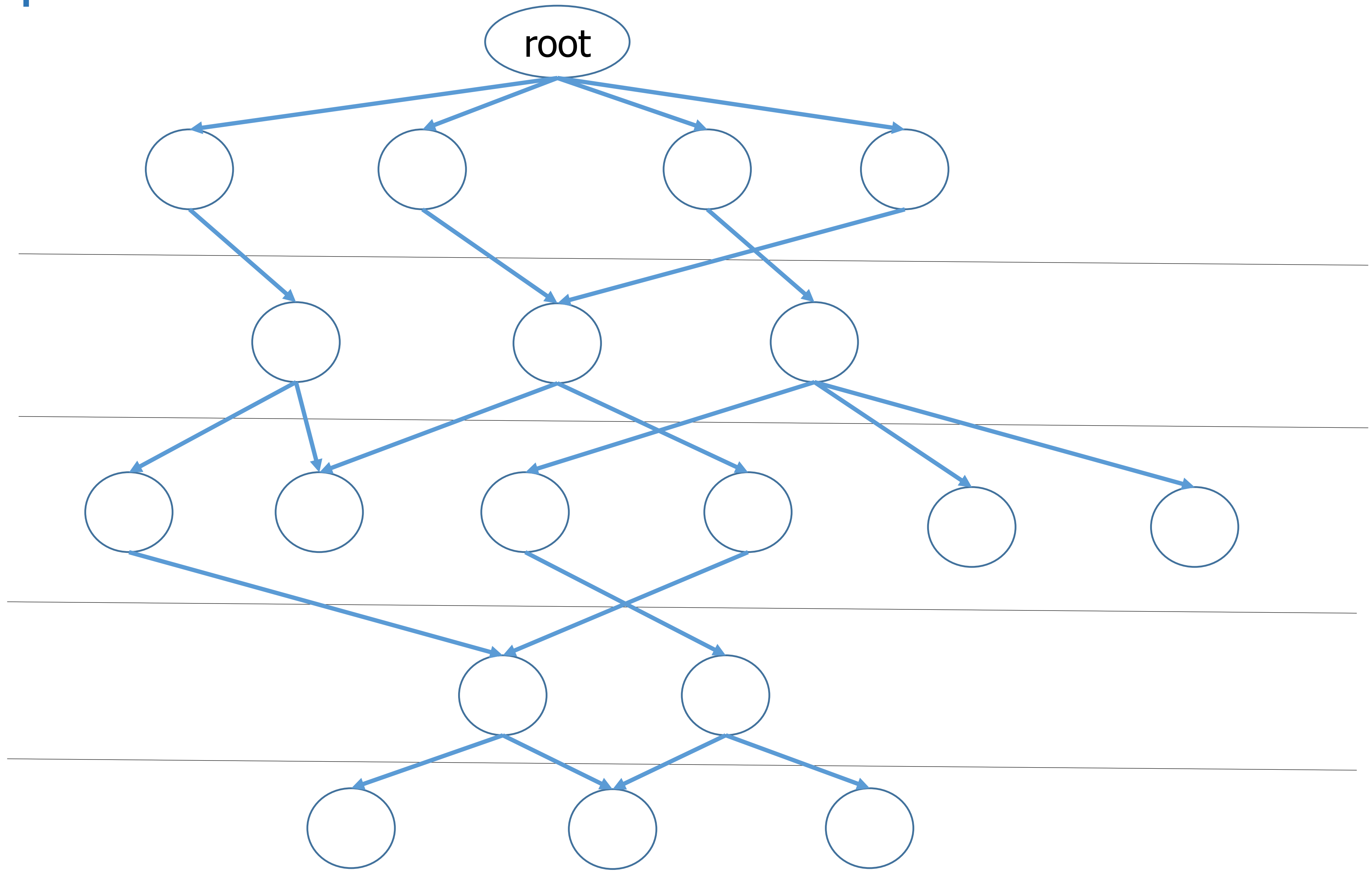
Main Observation

- Min-hop shortest path to u must pass through one of the neighbors v
- Min-hop shortest path to u can be found by extending the min-hop shortest path to one of its neighbors



Shortest path structure

MinHop



Algorithm

- We do not know the min-hop counts
- We will try all possibilities – only n choices are there.

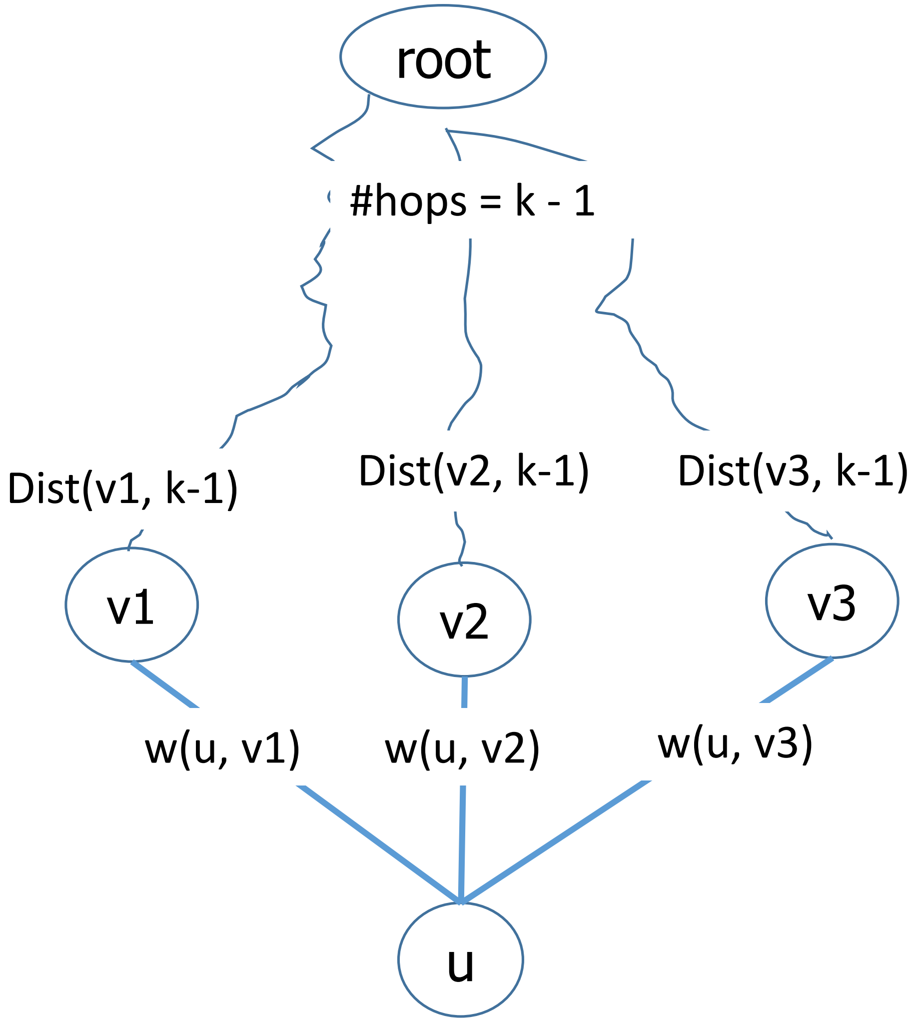
Generalize a bit

Dist(u, k) – shortest distance from root to u using at most k hops

Input:
- An undirected graph
- Weights on the edges
- A root vertex
- Number of allowed hops k

Output:
- Shortest path from the root to all the other vertices using at most k hops

Recurrence Relation



$$\boxed{Dist(u,k) = \min_{\substack{neighbors \\ v}} Dist(v,k-1) + w(u,v)}$$

- At most k hops
- We are allowed to use lesser number of hops

$$Dist(u,k) = \mathrm{Min}\left[\begin{array}{l} Dist(u,k-1) \\ \min_{\substack{neighbors \\ v}} Dist(v,k-1) + w(u,v) \end{array}\right.$$

Algorithm

```
For k = 1 to n                      Guess for min-hops
    For each u                      Compute Dist(u, k) for all u
        Dist(u, k) = Dist(u, k-1)   Current best distance
        For each neighbor v         Try all neighbors
            d = Dist(v, k-1) + w(u,v)      Best distance going via v
            If d is smaller than Dist(u, k)    Is this better?
                Dist(u, k) = d      If so, take it
```

- k can take at most n choices
- u can take at most n choices
- v can take at most n choices
- Running time O(n^3)

MinDist(root, u) = Dist(u, n)

Theorem: Our algorithm finds the shortest paths. Its running time is at most O(n^3)

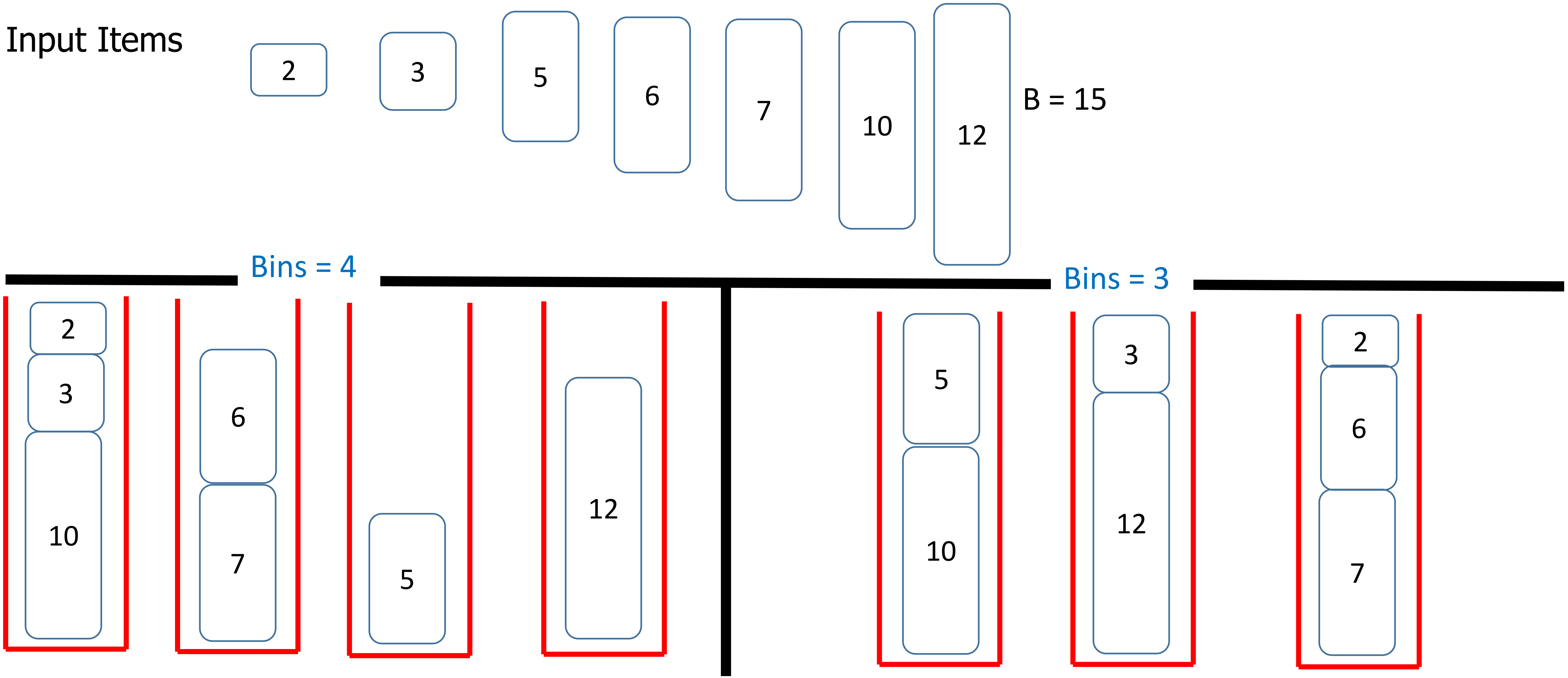
Bin Packing Problem

Bin Packing Problem

- Input : A set of items each having size
  Bins each of capacity B
- Objective : Pack the items in as few bins as possible

Have we seen this before?

- B = Sum / 2
- Allow only two bins
- What is this?



How many choices?

Naïve method:

- Try all possible ways to partition the set
- For each choice, check if it is feasible
- Choose the one having the least number of parts

Input Items: 2 3 5 6 7 10 12

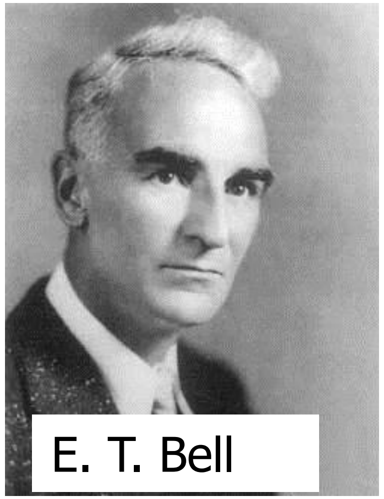
Partition 1: [2 3 6 5] [7 10 12] — Not feasible

Partition 2: [2 5] [6 3 7] [10 12] — Not feasible

Partition 3: [2 5] [6 7] [10] [12 3] — Feasible, Bins = 4

Number of partitions

- Bell number ≈ n^n



E. T. Bell

What can we aim for?

- NP-hard : we can find the optimal solution in polynomial time
- Naïve algorithm: runs in roughly O(n^n)
- Our algorithm: O(4^n)

Recursion

- Input : A set of n items

Key Idea : Guess the items that go into the first bin and then recurse

Guess 1 2 3 4 5 6 7

1 3 4 | 2 5 6 7

First bin

Other bins
Solve recursively

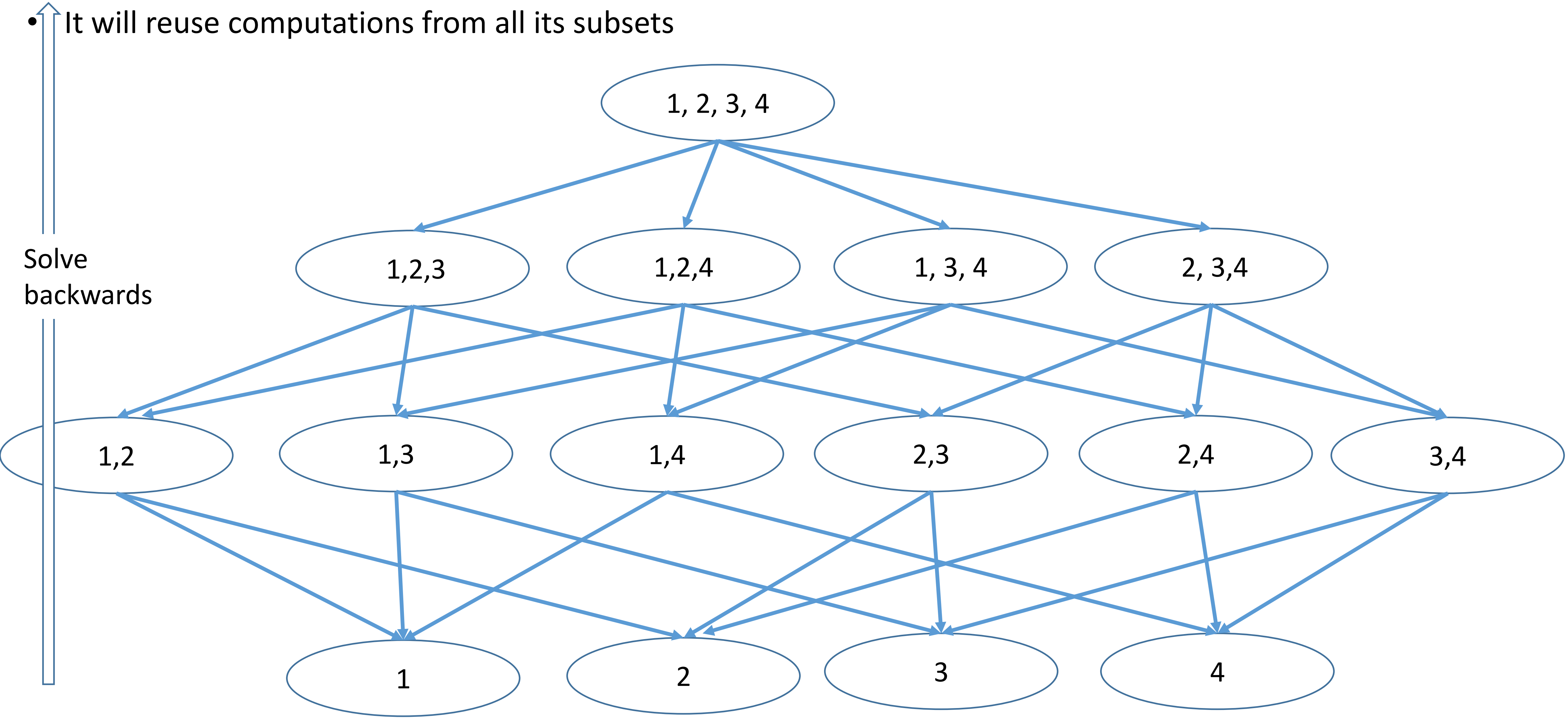
Make sure this is feasible : size < B

$$\text{MinBins}(I) = 1 + \min_{\substack{S \subseteq I \\ S \text{ is feasible}}} \left[\text{MinBins}(S) \right]$$

First bin used for S

Sub-problem structure

- Each subset is a sub-problem
- It will reuse computations from all its subsets



Algorithm

```
For k = 1 to n                          Size of sets
    For each set X of size k            Each set of size k
        MinBins [X] = infity            Initialize
        For each subset S of X          Guess for first bin
            b = 1 + MinBins[X – S]      Look up table
            if(b < MinBins [X] )        Is this better
                MinBins[X] = b          If so take it
```

- Number of possible X : 2^n
- Number of possible S : 2^n
- Total running time : 4^n

Theorem: Our algorithm finds the optimal solution. Its running time is at most O(4^n)

That’s about it!

Have a nice time designing dynamic programs!