DTS203TC
Design and Analysis of Algorithms

Lecture 11: Greedy Algorithms

Dr. Qi Chen
School of AI and Advanced Computing

Learning outcome

- Understand what greedy algorithm is
- Able to apply greedy algorithm to solve
  - the Activity Selection problem
  - the Huffman Coding problem
- Able to apply greedy algorithm to find solution for Knapsack problem

Coin Change Problem

Suppose we have 3 types of coins

0.1

0.5

1.0

Minimum number of coins to make
0.8, 1.0, 1.4 ?

Greedy method

Greedy Algorithms

How to be greedy?

- At every step, make the best move you can make
- Keep going until you're done

Advantages

- Don't need to pay much effort at each step
- Usually finds a solution very **quickly**
- The solution found is usually **not bad**

Possible problem

- The solution found may **NOT** be the best one

Greedy methods - examples

- Two examples of greedy algorithms:
  - Activity Selection
  - Huffman Coding
- One **non**-example of a greedy algorithm:
  - Knapsack
- We will cover other examples later:
  - Minimum spanning tree
  - Shortest paths

Activity Selection

Programming

DTS204TC Class

DTS203TC Lab

Sleep

You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

DTS203TC Class

DTS205TC Class

Watch Movies

Seminar

Lunch

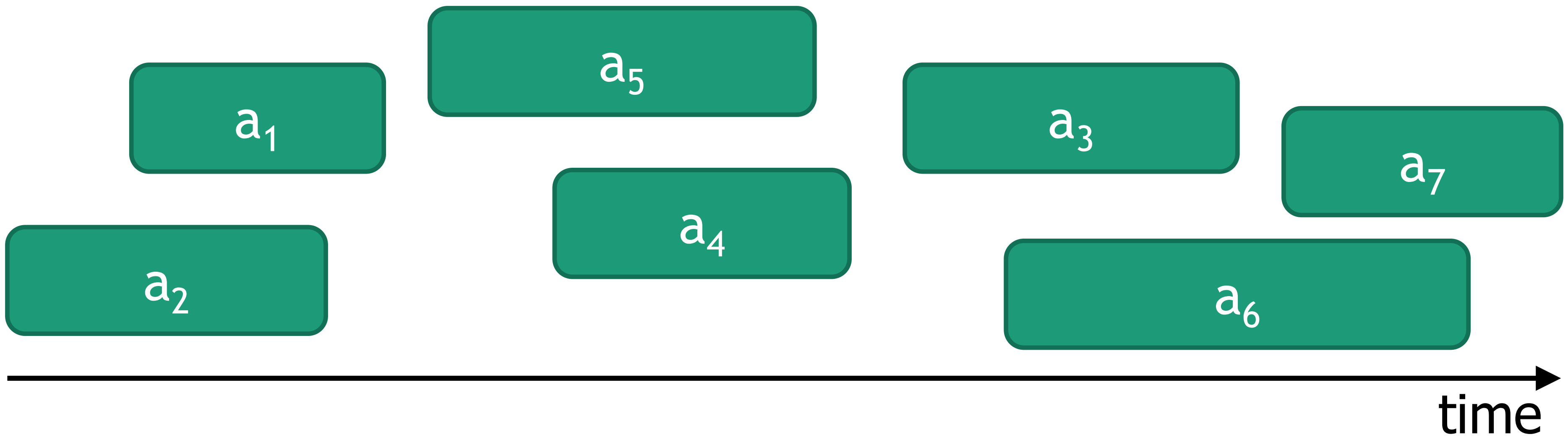
Social activity

time

Activity selection

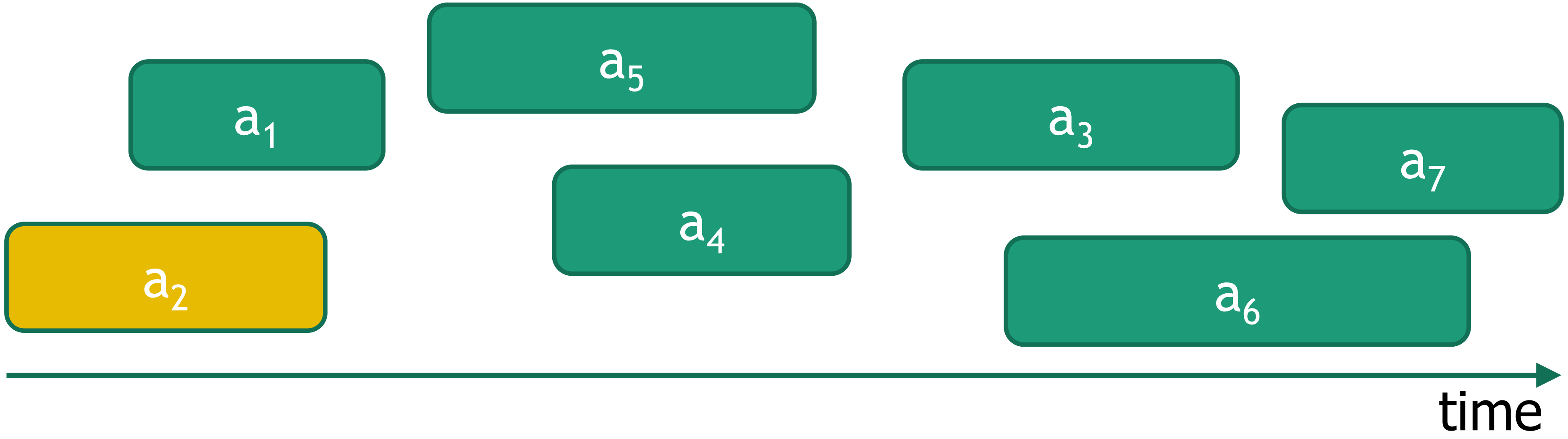
- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$
- Output:
  - How many activities can you do today?

Greedy Algorithm



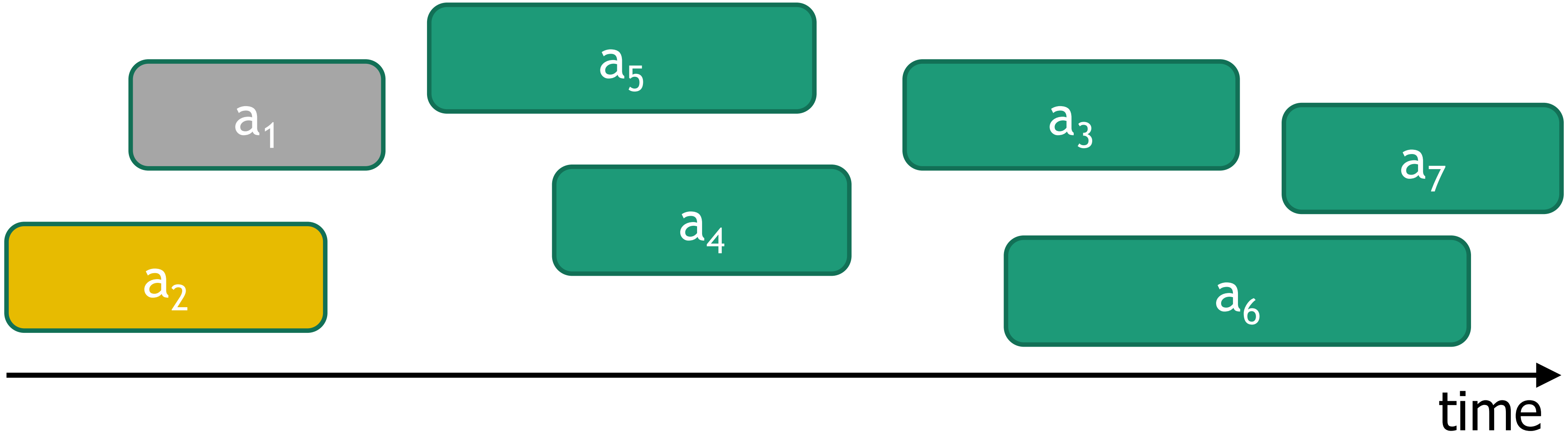
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



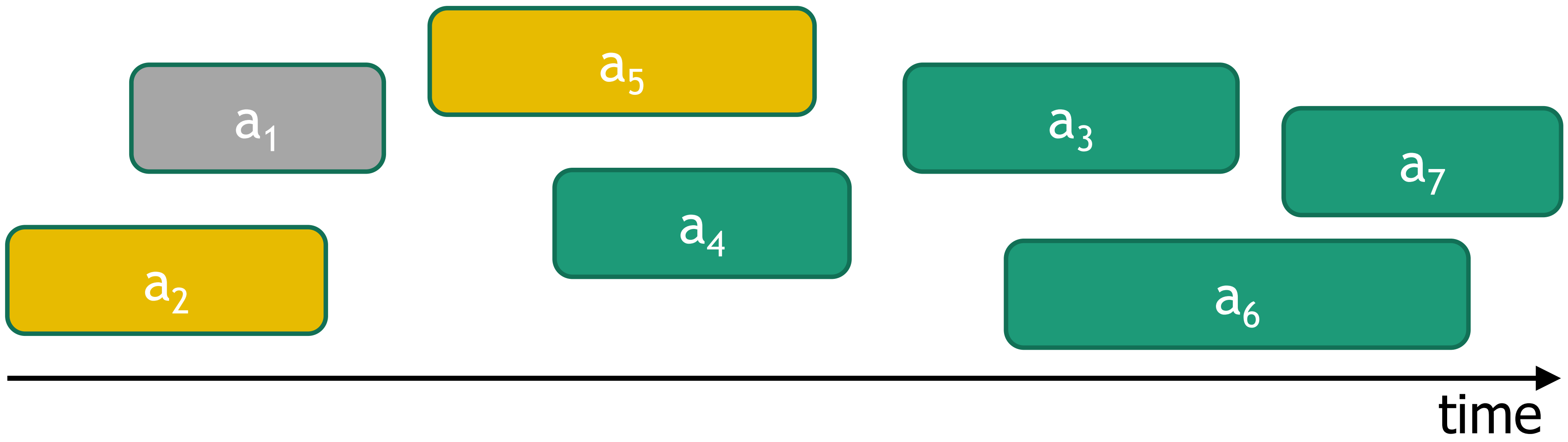
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



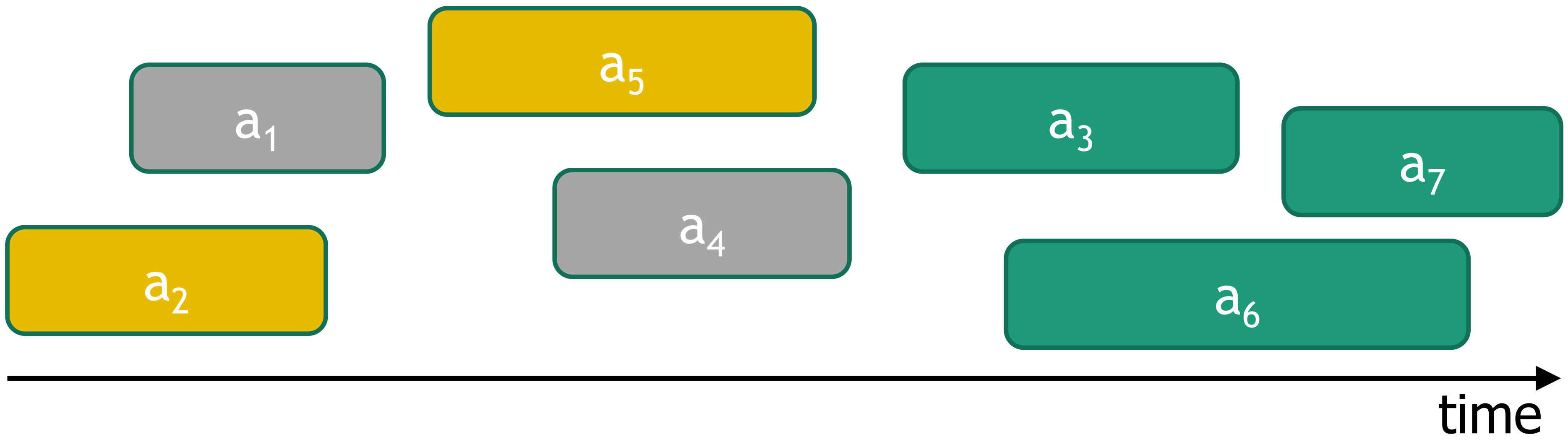
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



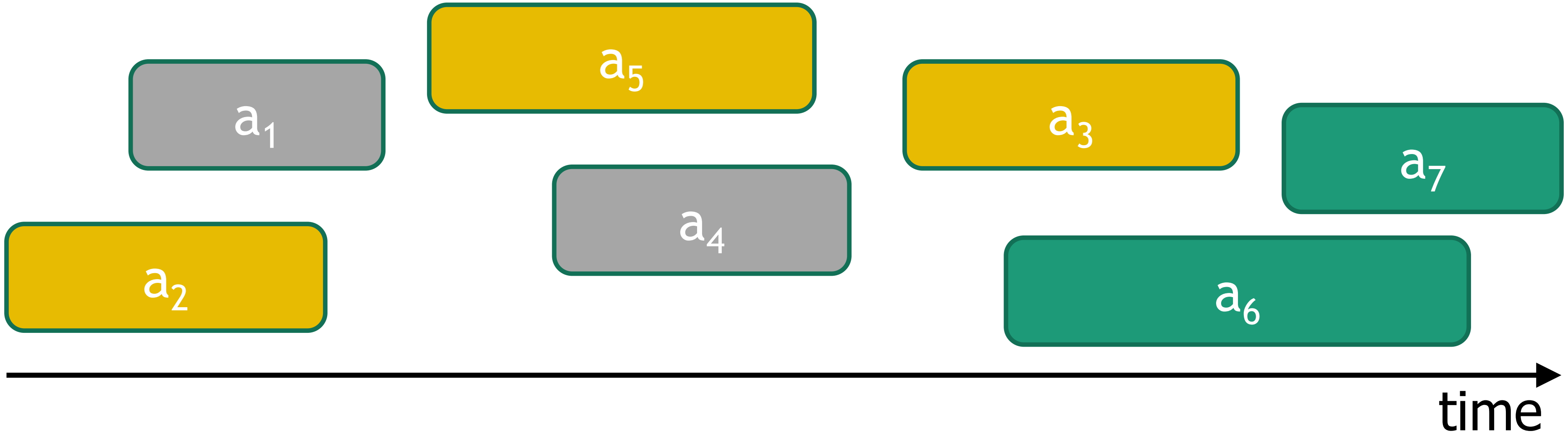
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



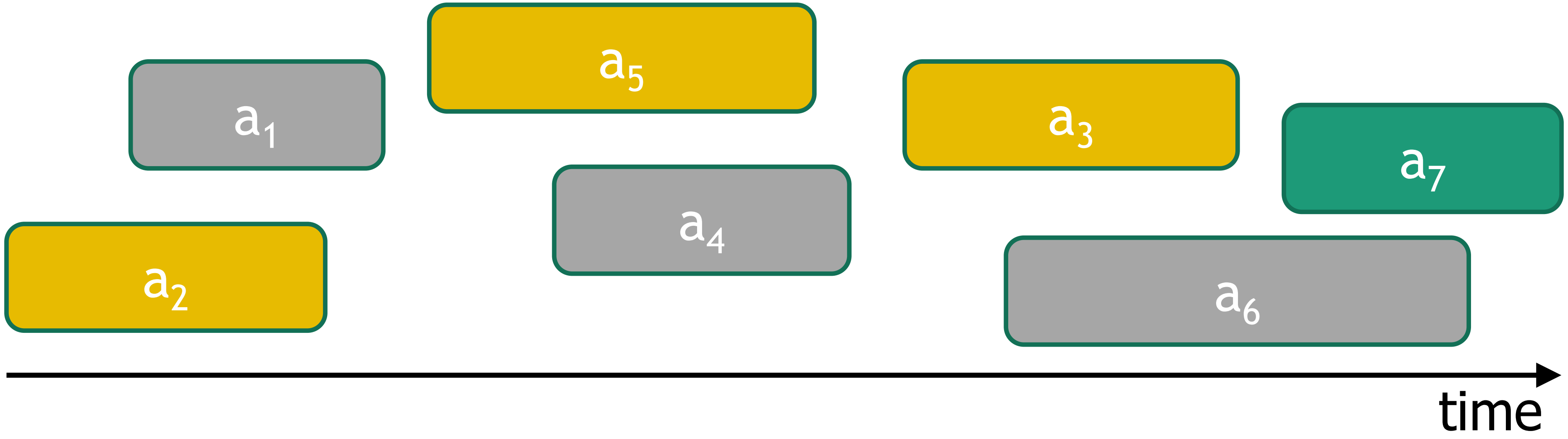
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



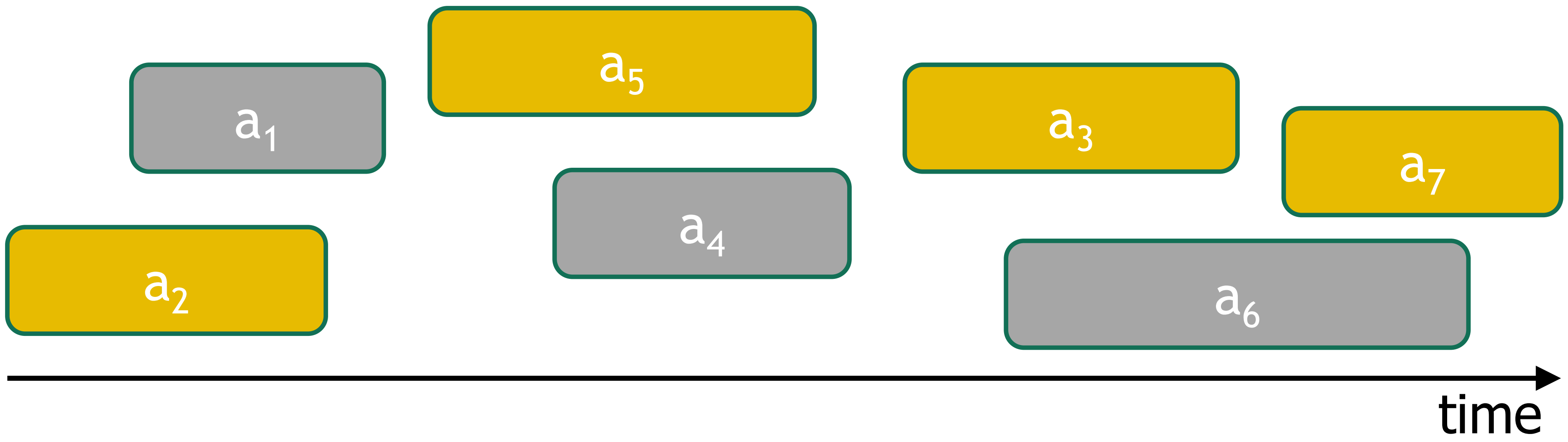
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

At least it's fast

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise O(nlog(n)) if you have to sort them first.

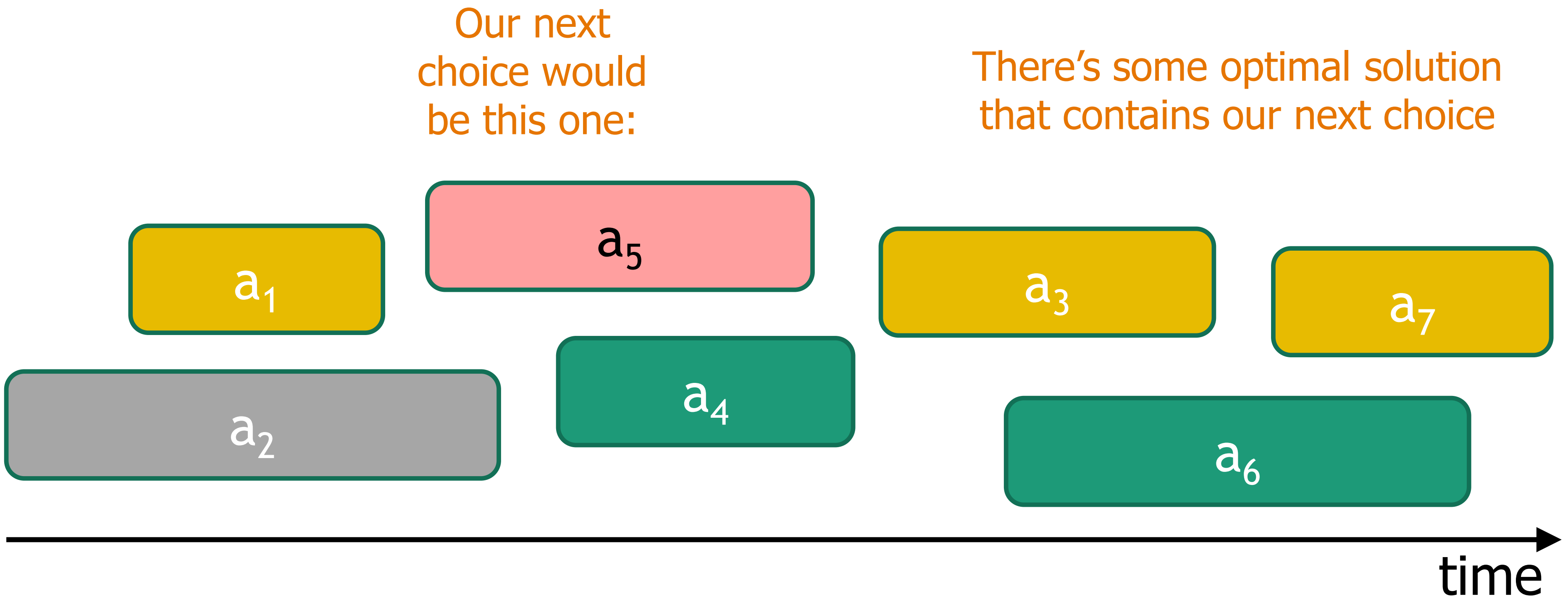
What makes it greedy?

- At each step in the algorithm, make a choice.
  - Increase my activity set by one,
  - Leave lots of room for future choices,
  - Repeat and hope for the best!

- **Hope** that at the end of the day, this results in a globally optimal solution.

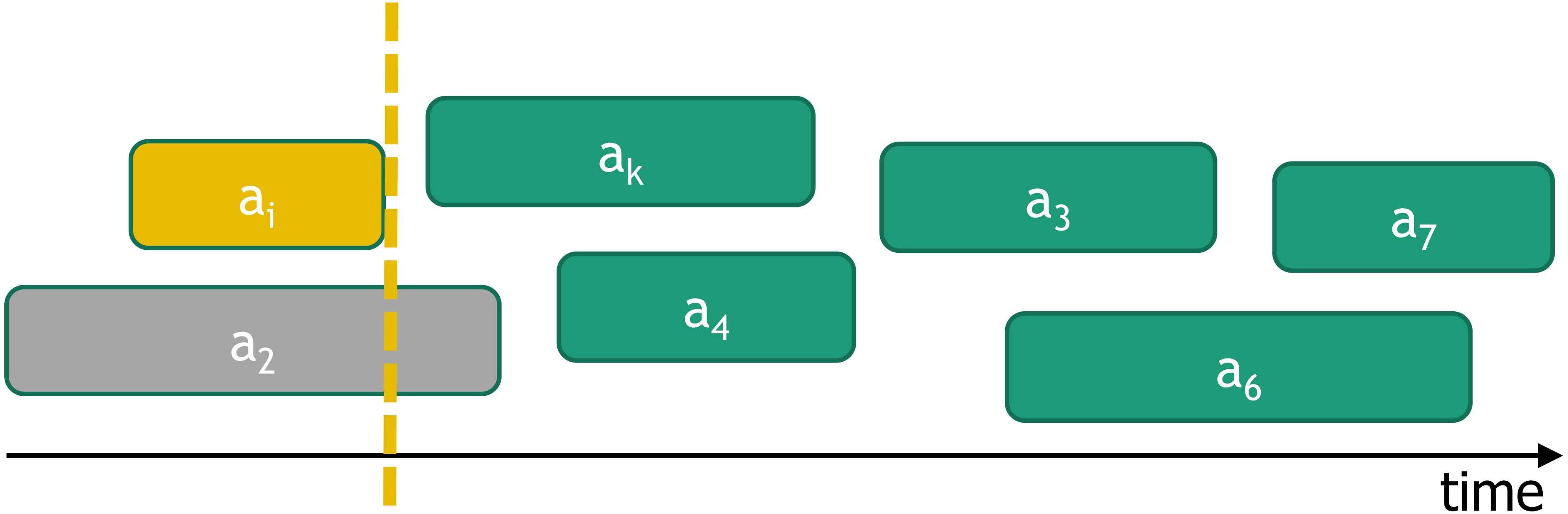
Why does it work?

- Whenever we make a choice, **we don't rule out an optimal solution.**



Optimal Substructure

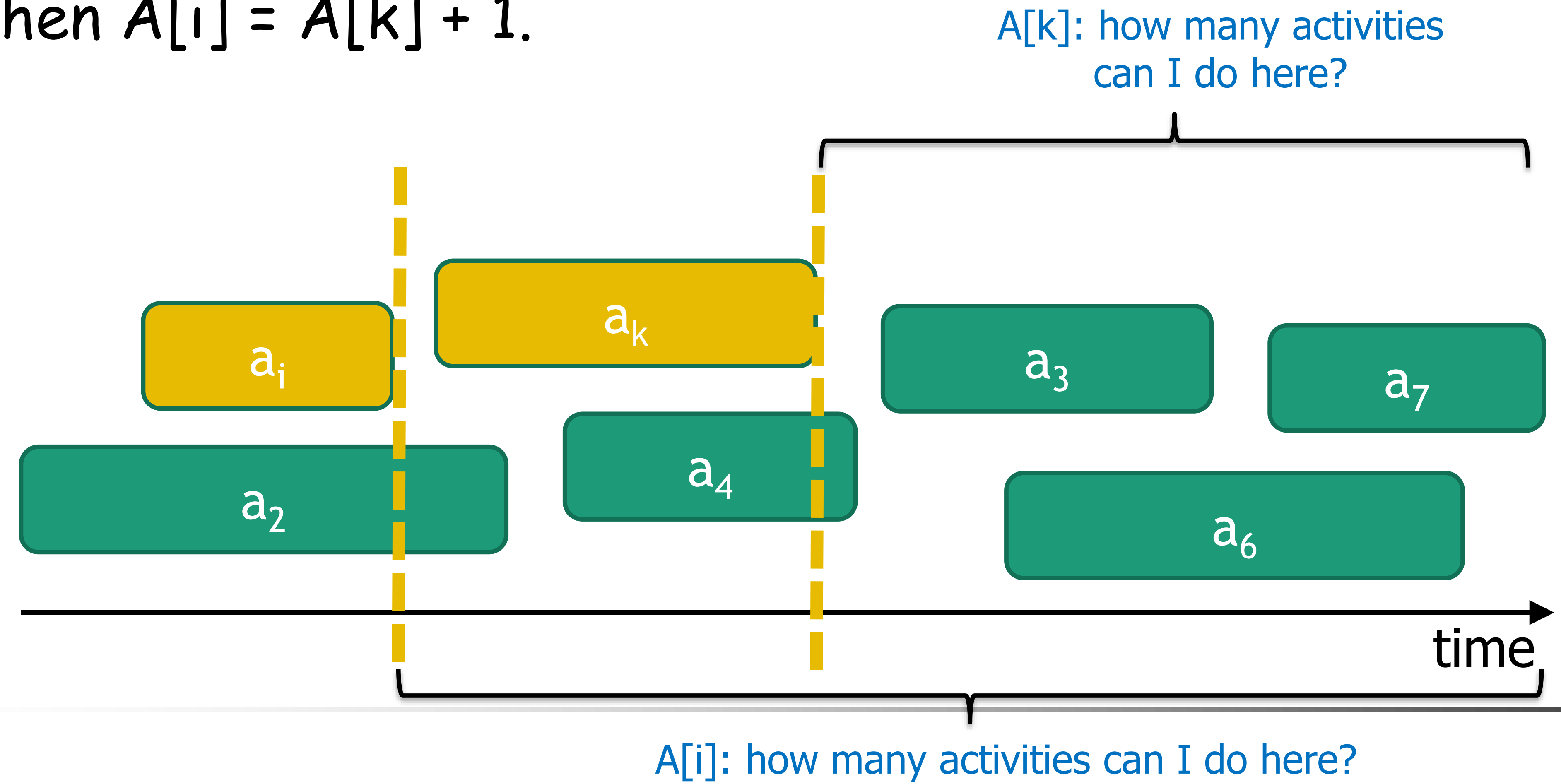
- Subprogram i :
  - A[i] = Number of activities you can do after Activity i finishes.



Want to show: when we make a choice a_k, the optimal solution to the smaller sub-problem k will help us solve sub-problem i

Optimal Substructure

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then $A[i] = A[k] + 1$.



Optimal Substructure

- If we choose a_k have the smallest finish time among activities do-able after a_i finishes, then A[i] = A[k] + 1.

- That is:
  - Assume that we have an optimal solution up to a_i
  - By adding a_k we are still on track to hit that optimal value

Common strategy

- Make a series of choices.
- Show that, at each step, our choice **won't rule out an optimal solution.**
- After we've made all our choices, we haven't ruled out an optimal solution, **so we must have found one.**

Huffman coding

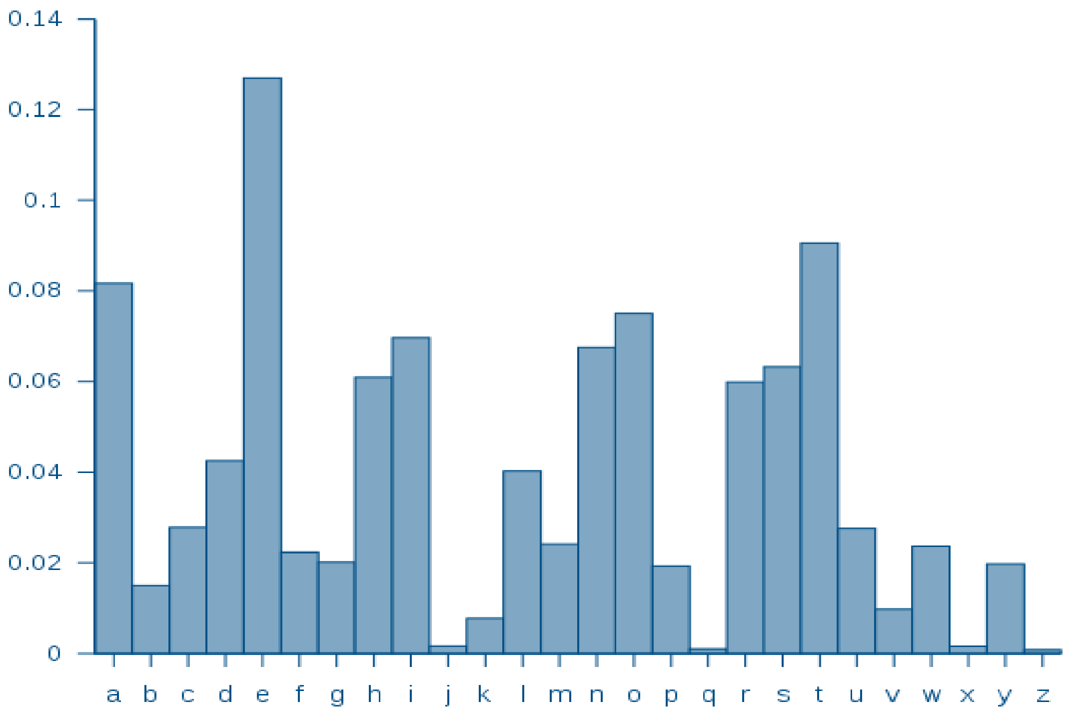
Huffman coding

- everyday english sentence
  - 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000 01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101
- qwertyui_opasdfg+hjklzxcv
  - 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Huffman coding

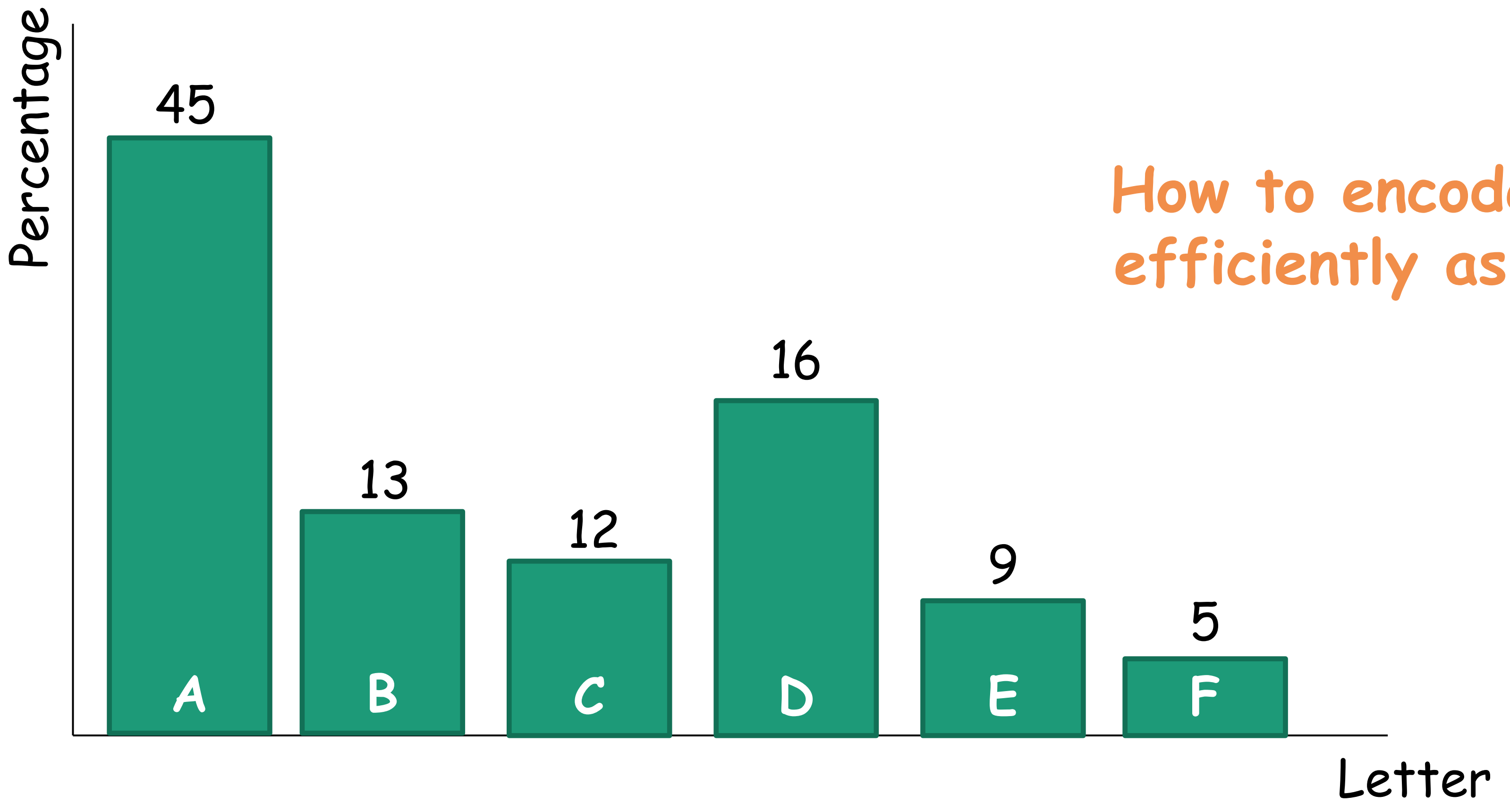
ASCII is pretty wasteful. If **e** shows up so often, we should have a more parsimonious way of representing it!

- everyday english sentence
  - 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000 01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101



Huffman coding

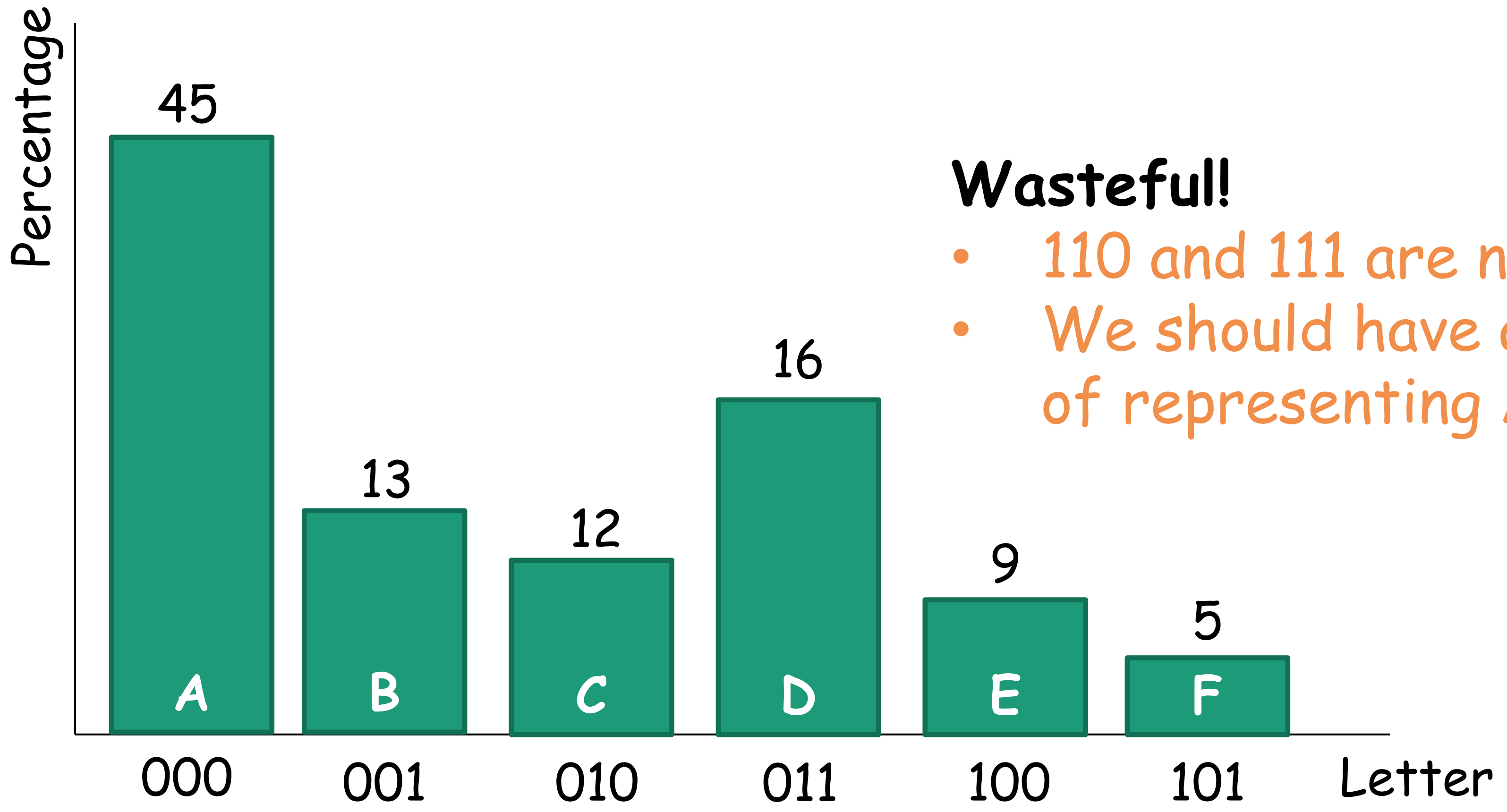
- Suppose we have some distribution on characters



How to encode them as efficiently as possible?

Try 0

- Every letter is assigned a **binary string** of three bits.

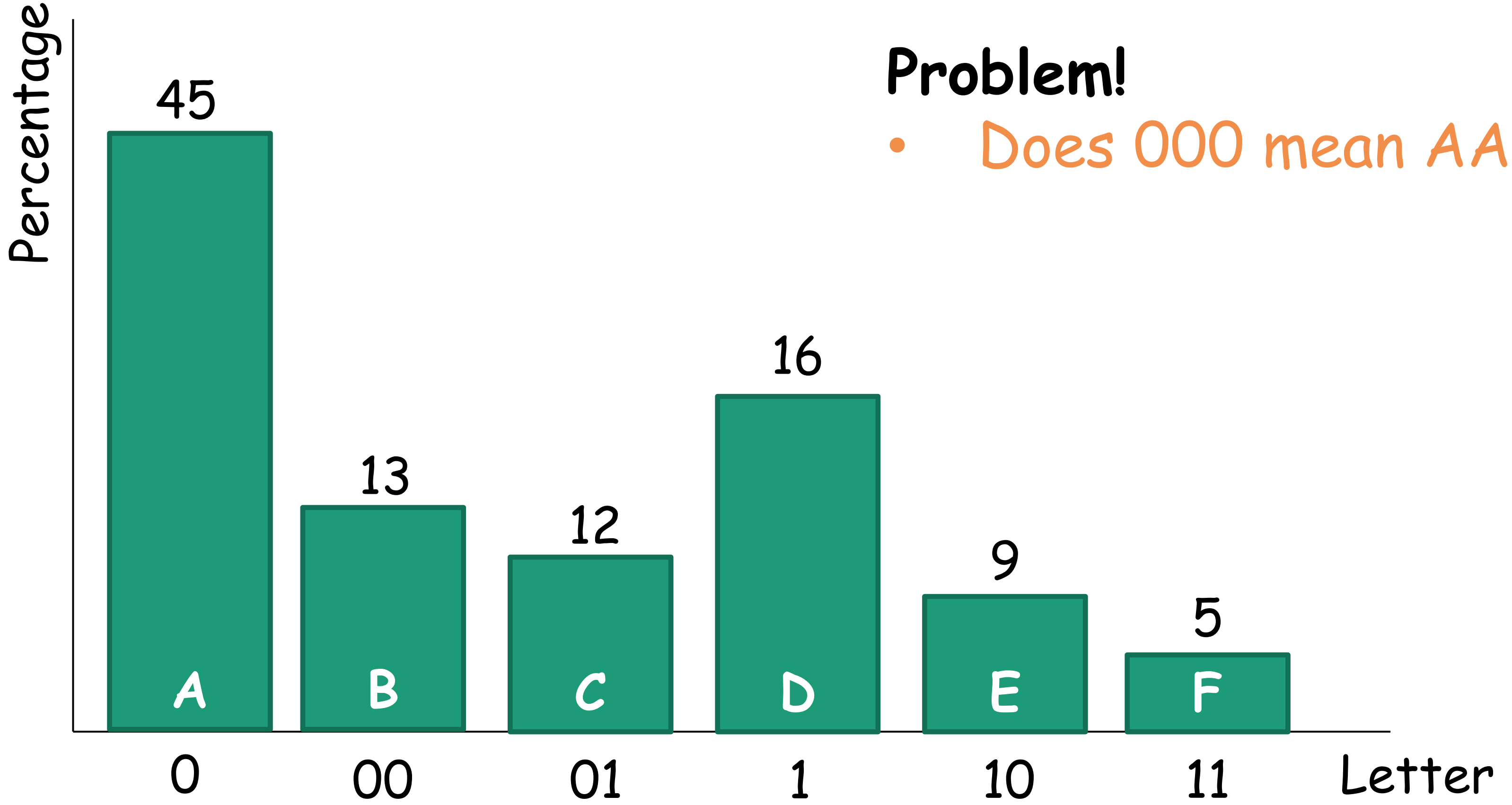


Wasteful!

- 110 and 111 are never used.
- We should have a shorter way of representing A.

Try 1

- Every letter is assigned a **binary string** of one or two bits.
- The more frequent letters get the shorter strings.



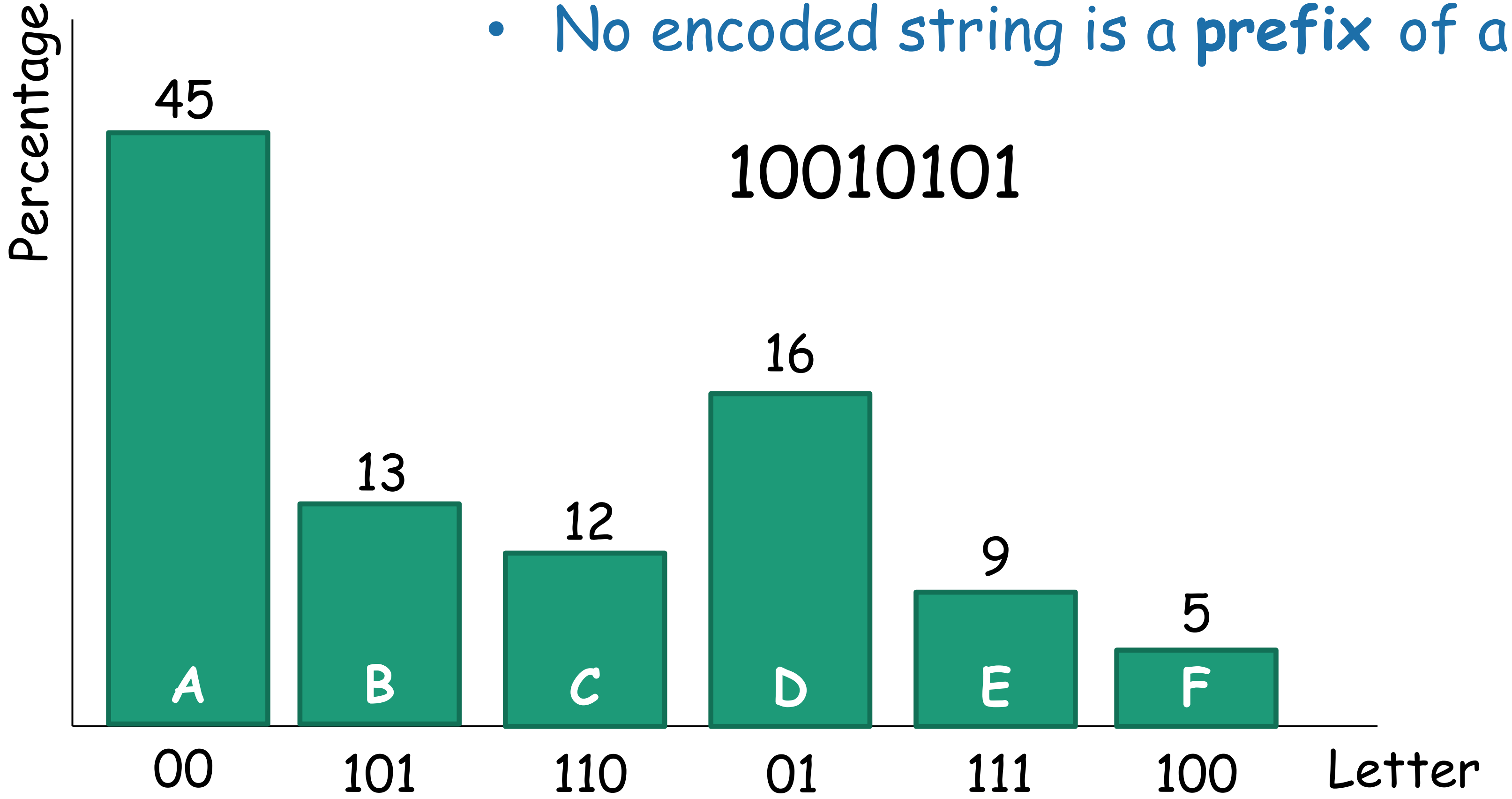
Problem!

- Does 000 mean AAA r BA or AB?

Try 2: Prefix codes

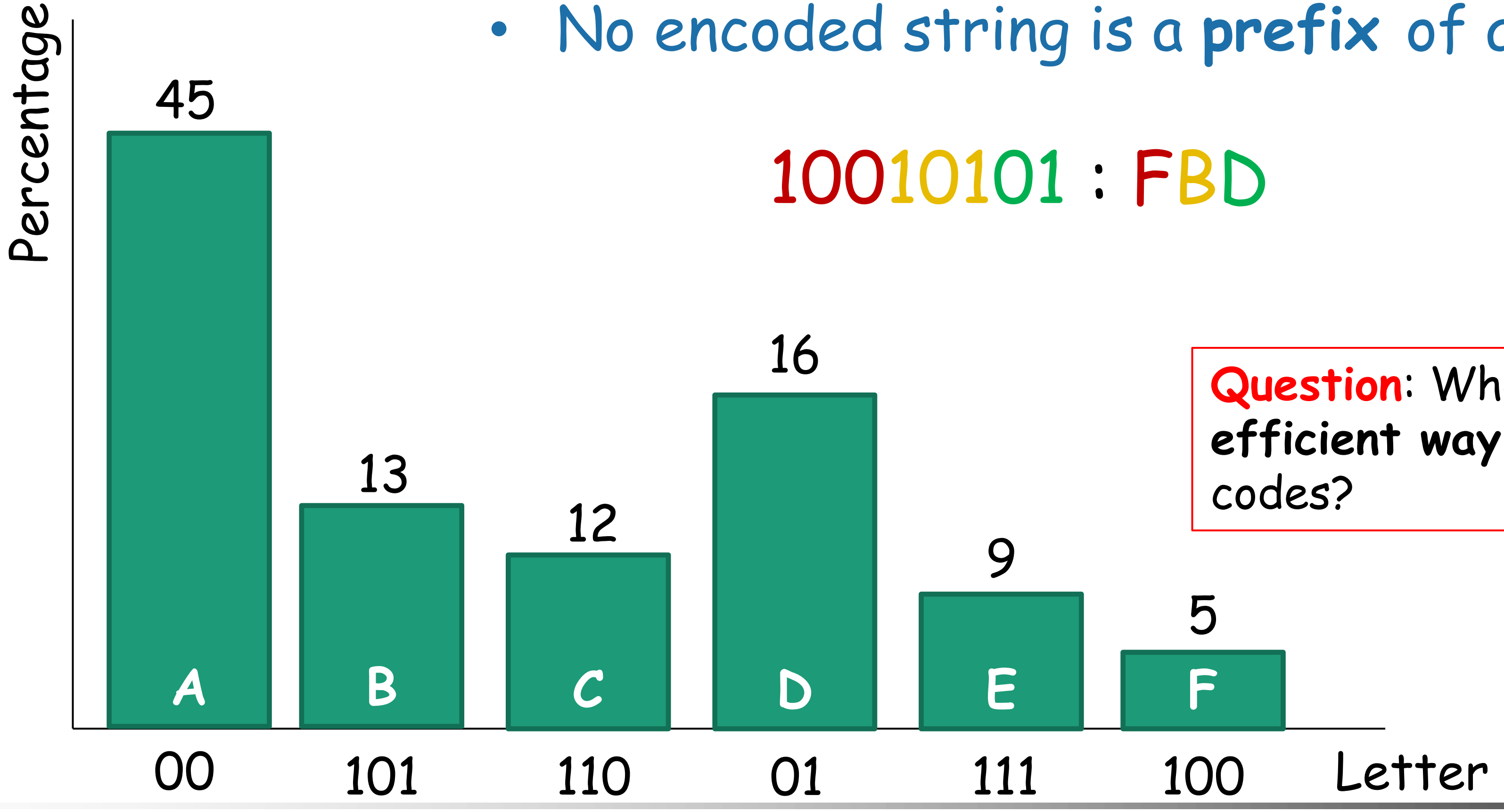
- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

10010101



Try 2: Prefix codes

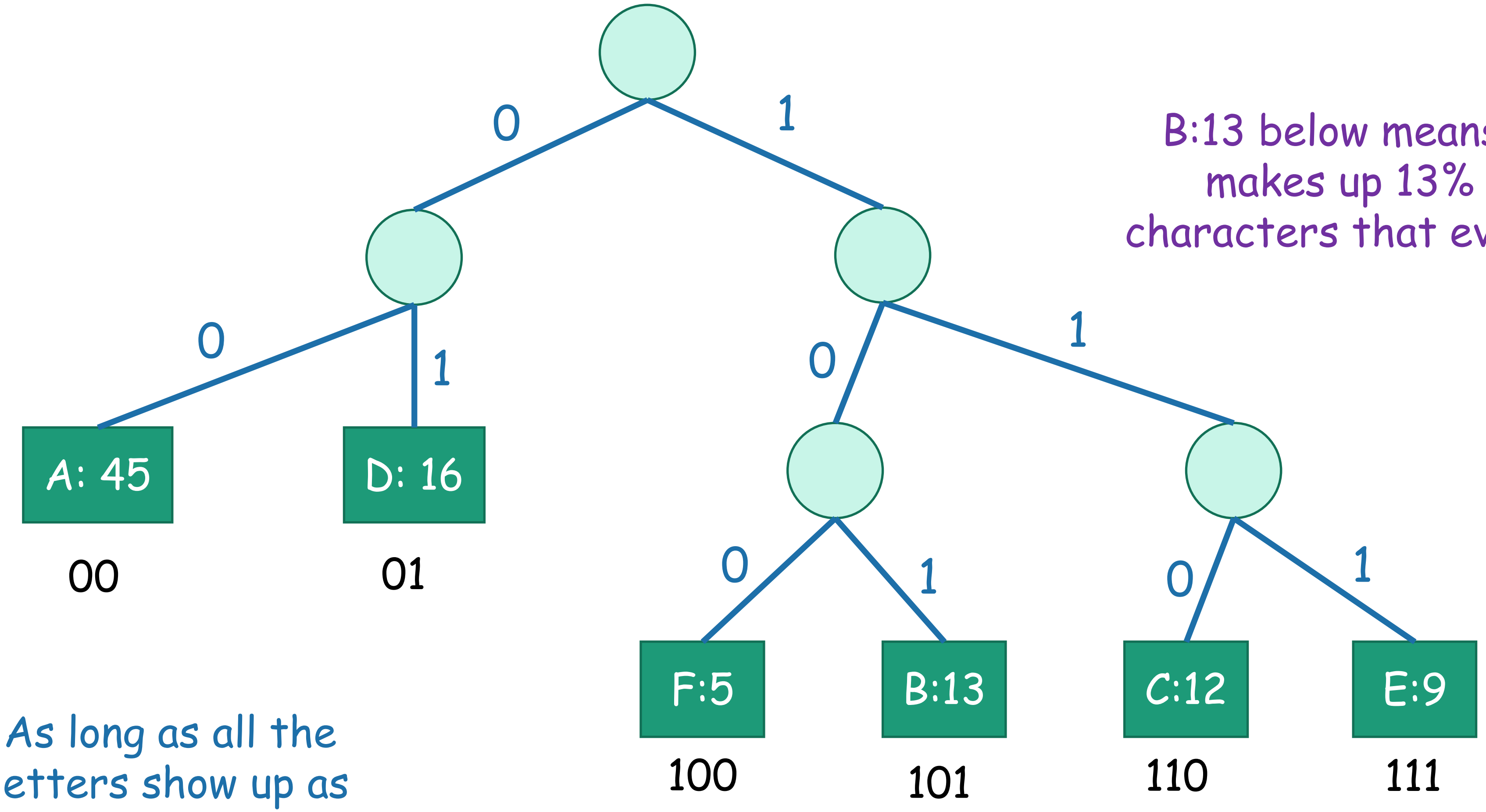
- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.



10010101 : FBD

Question: What is the most **efficient way** to find prefix codes?

A prefix code is a tree



B:13 below means that 'B' makes up 13% of the characters that ever appear.

As long as all the letters show up as leaves, this is prefix code.

Some trees are better than others

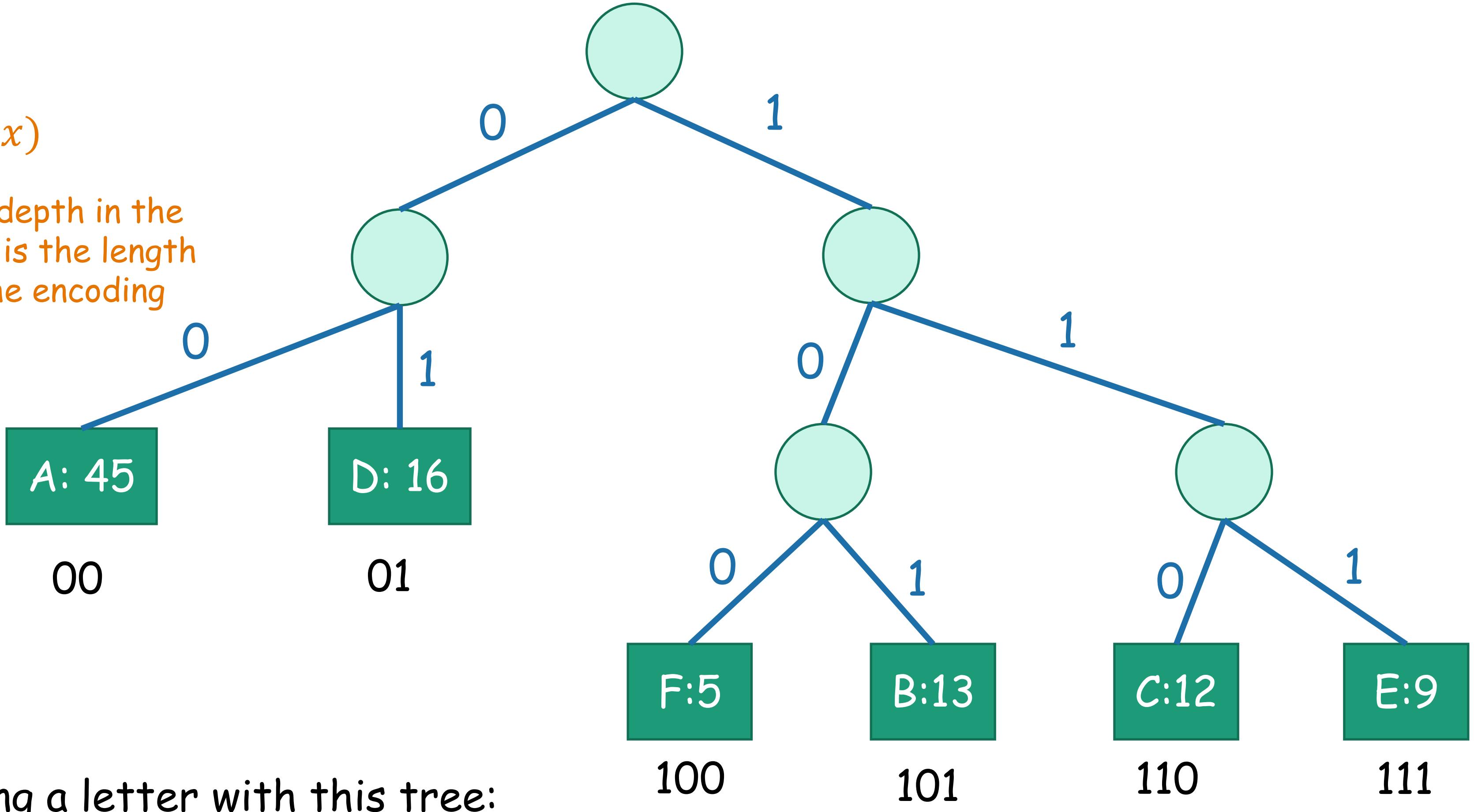
The **cost of a tree** is the expected length of the encoding of that letter.

Cost =

$$\sum_{leaves\ x} P(x) \cdot \text{depth}(x)$$

P(x) is the probability of letter x

The depth in the tree is the length of the encoding



Expected cost of encoding a letter with this tree:

$$\mathbf{2(0.45 + 0.16) + 3(\ 0.05 + 0.13 + 0.12 + 0.09) = 2.39}$$

Question

- Given a distribution P on letters, find the lowest-cost tree.

Cost =

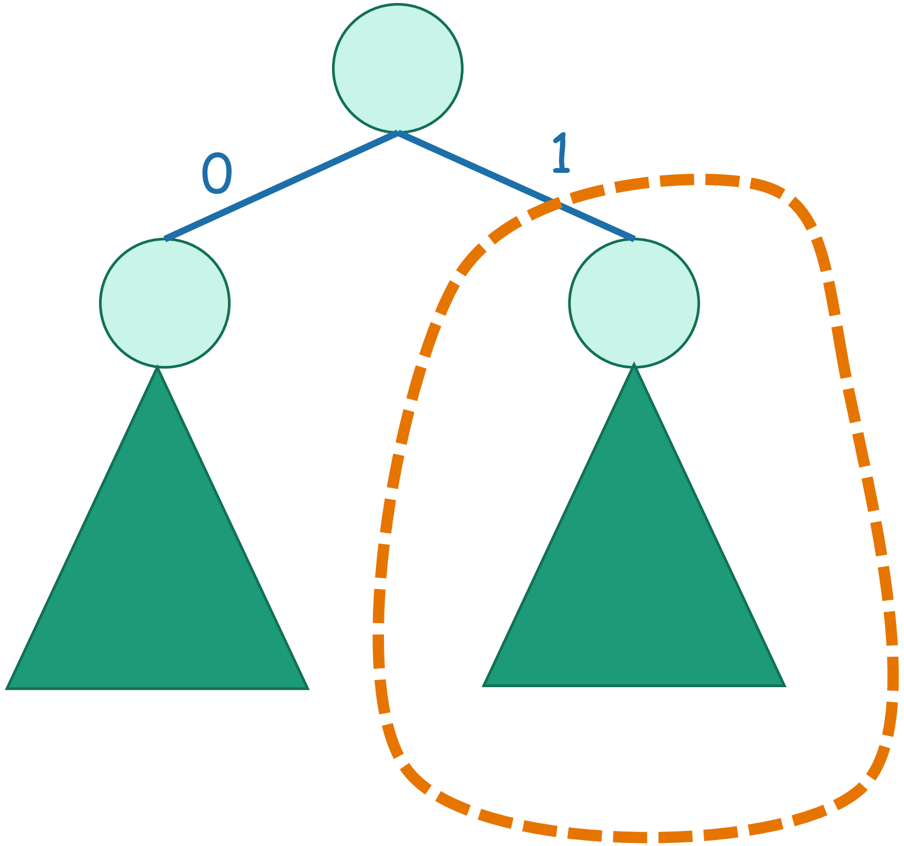
$$\sum_{leaves\ x} P(x) \cdot \text{depth}(x)$$

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

Optimal sub-structure

- Suppose this is an optimal tree:

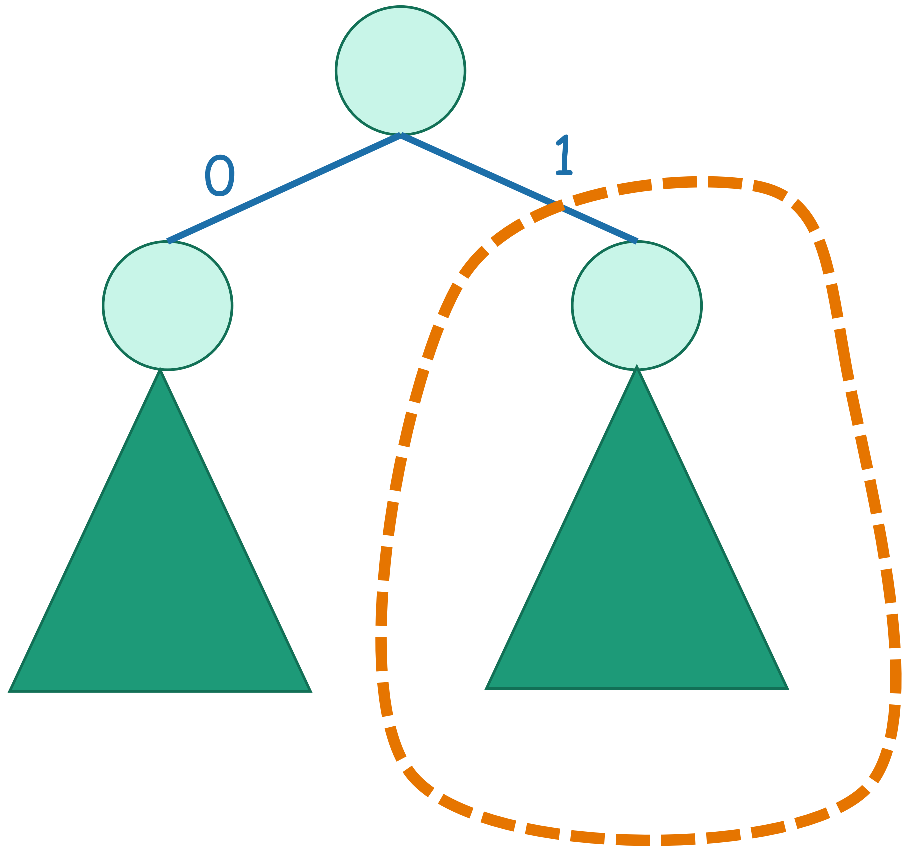


Then this is an optimal tree on fewer letters.

Otherwise, we could change this sub-tree and end up with a better overall tree.

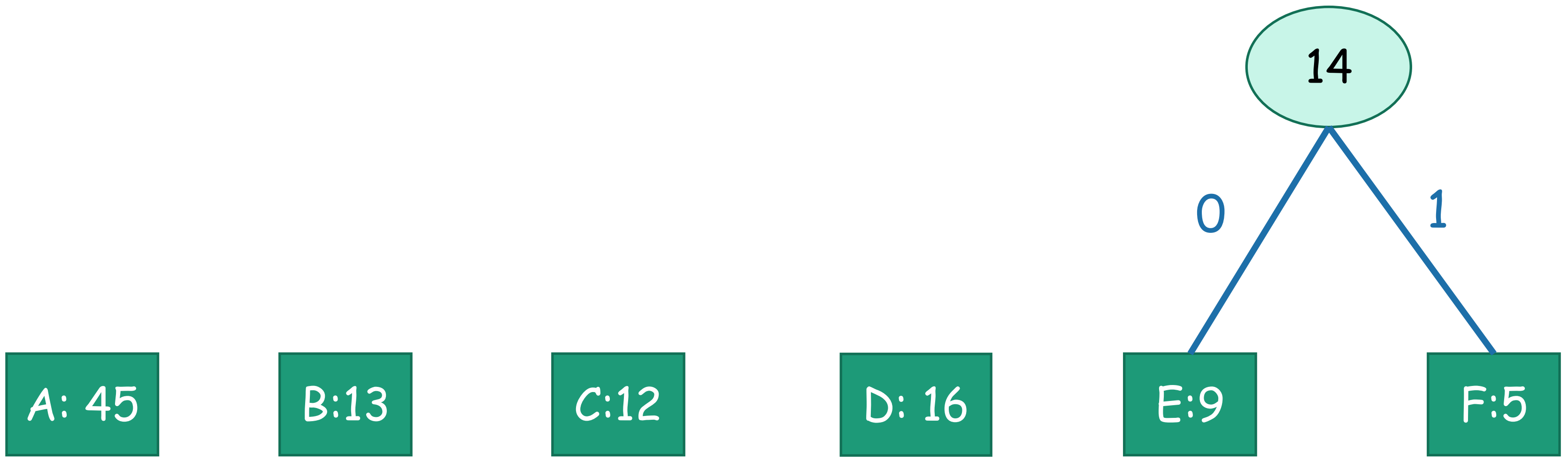
Optimal sub-structure

- Think about what letters belong in this sub-problem...

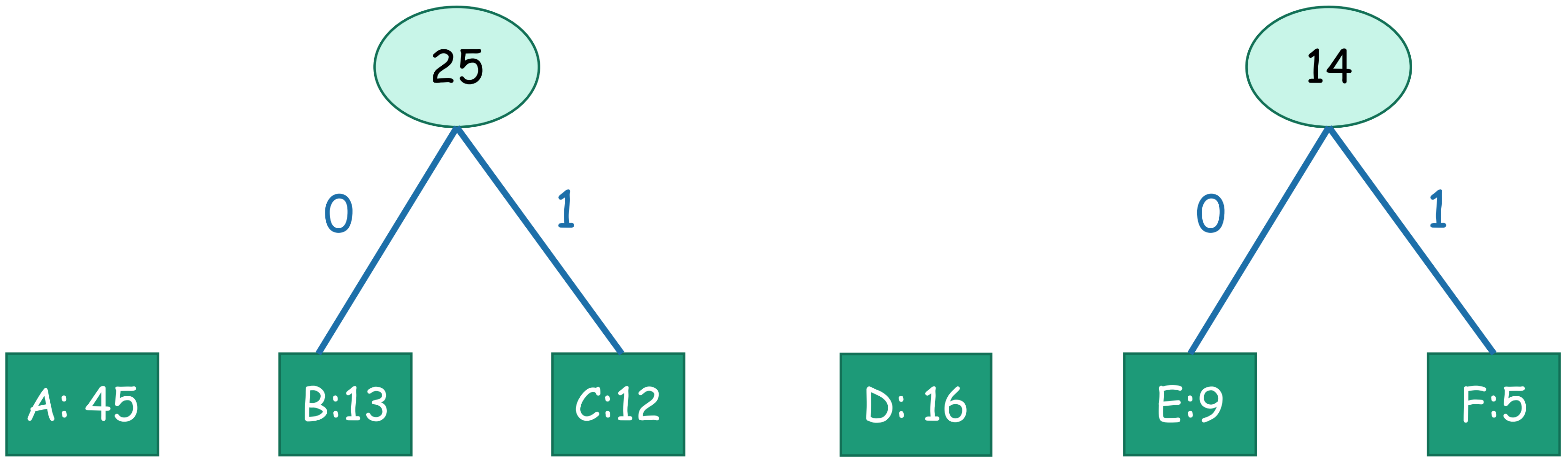


Infrequent elements!
We want them as low down as possible.

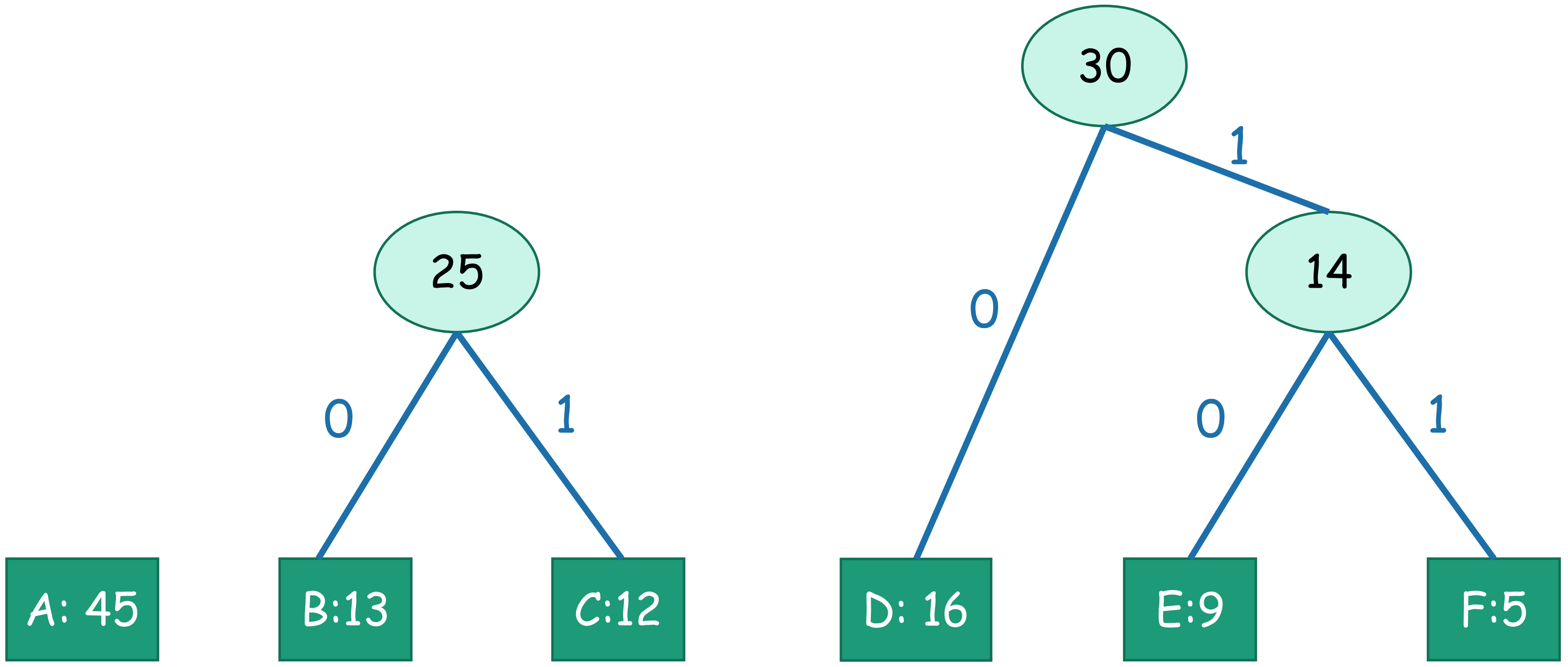
Solution



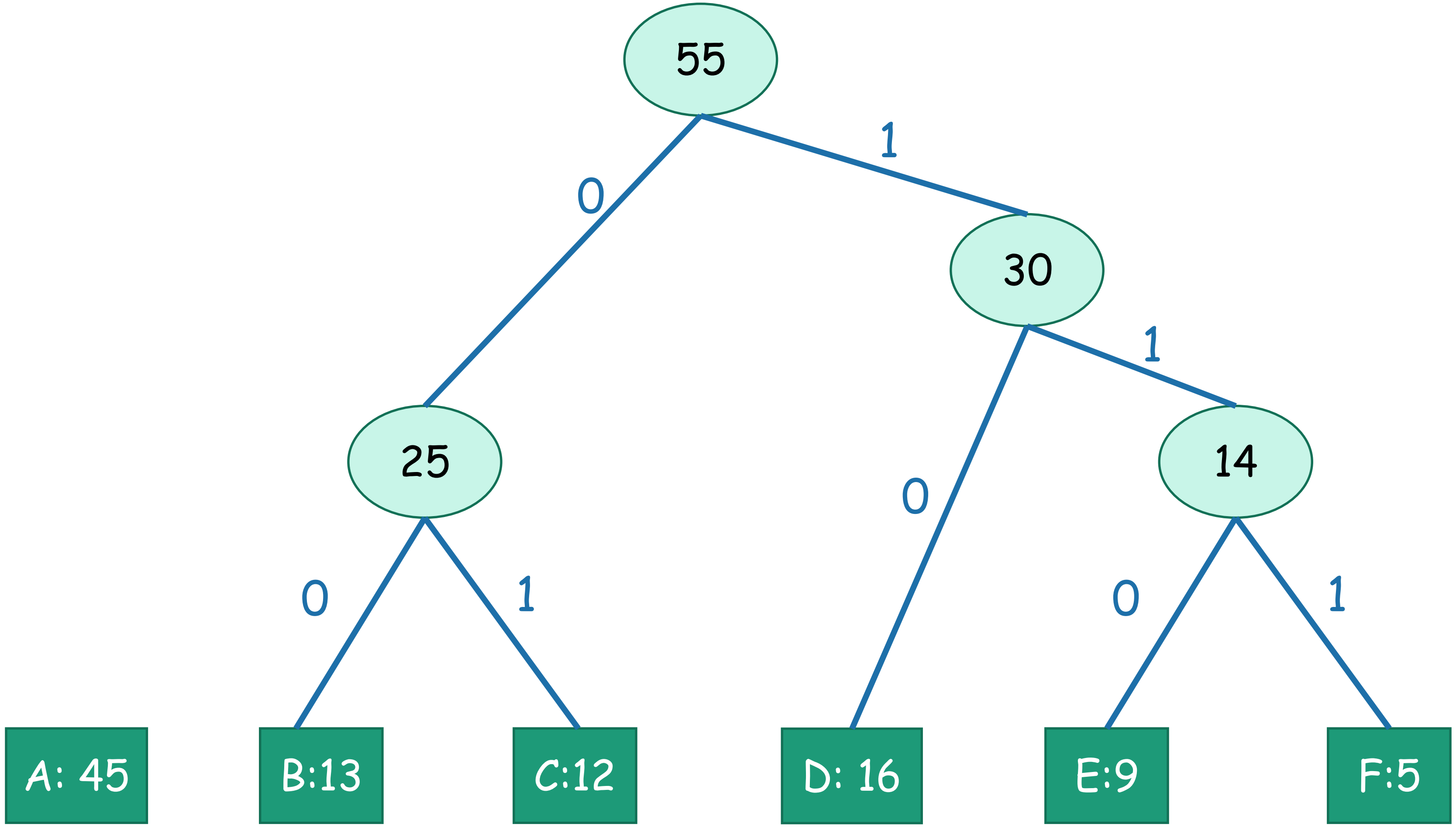
Solution



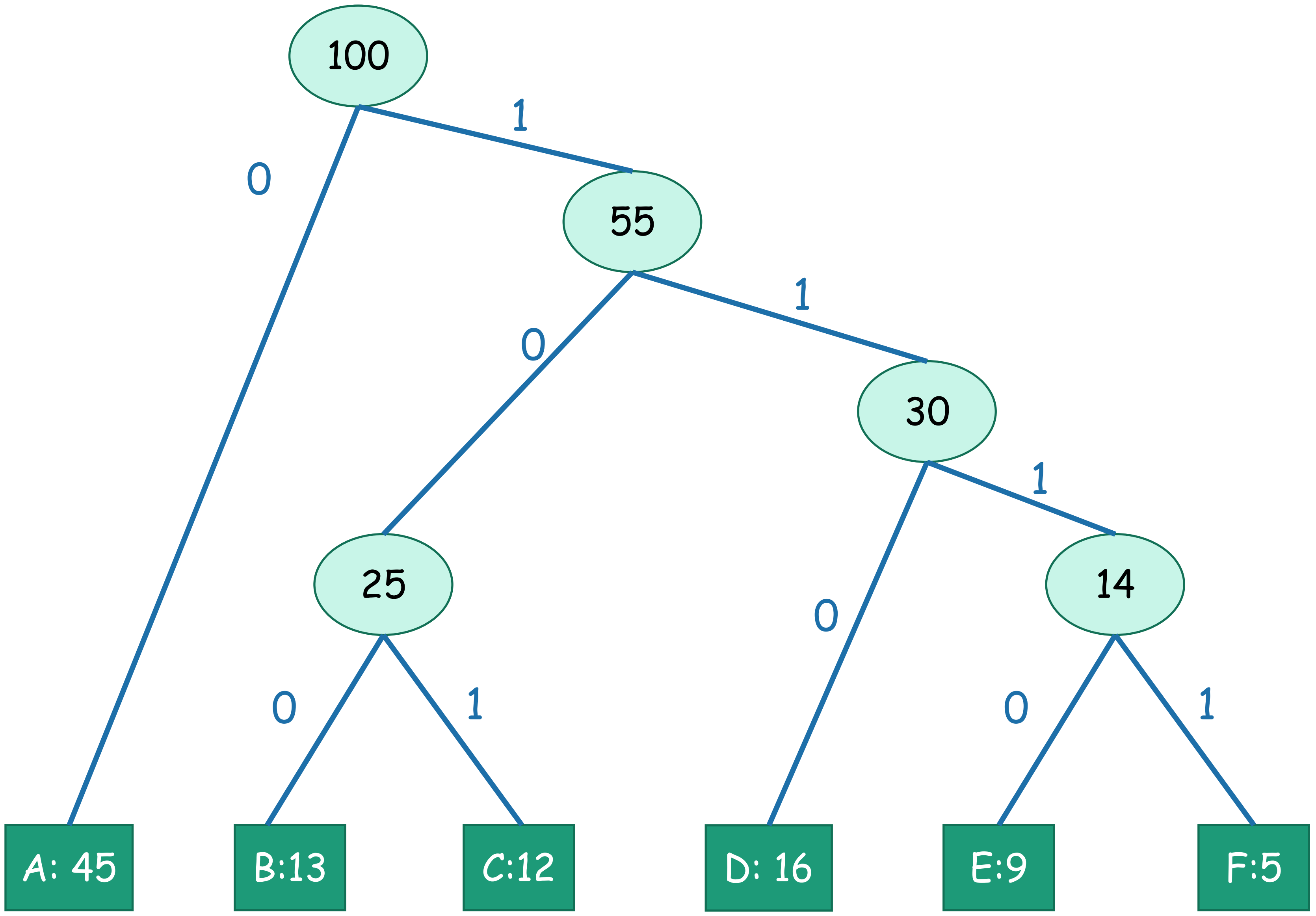
Solution



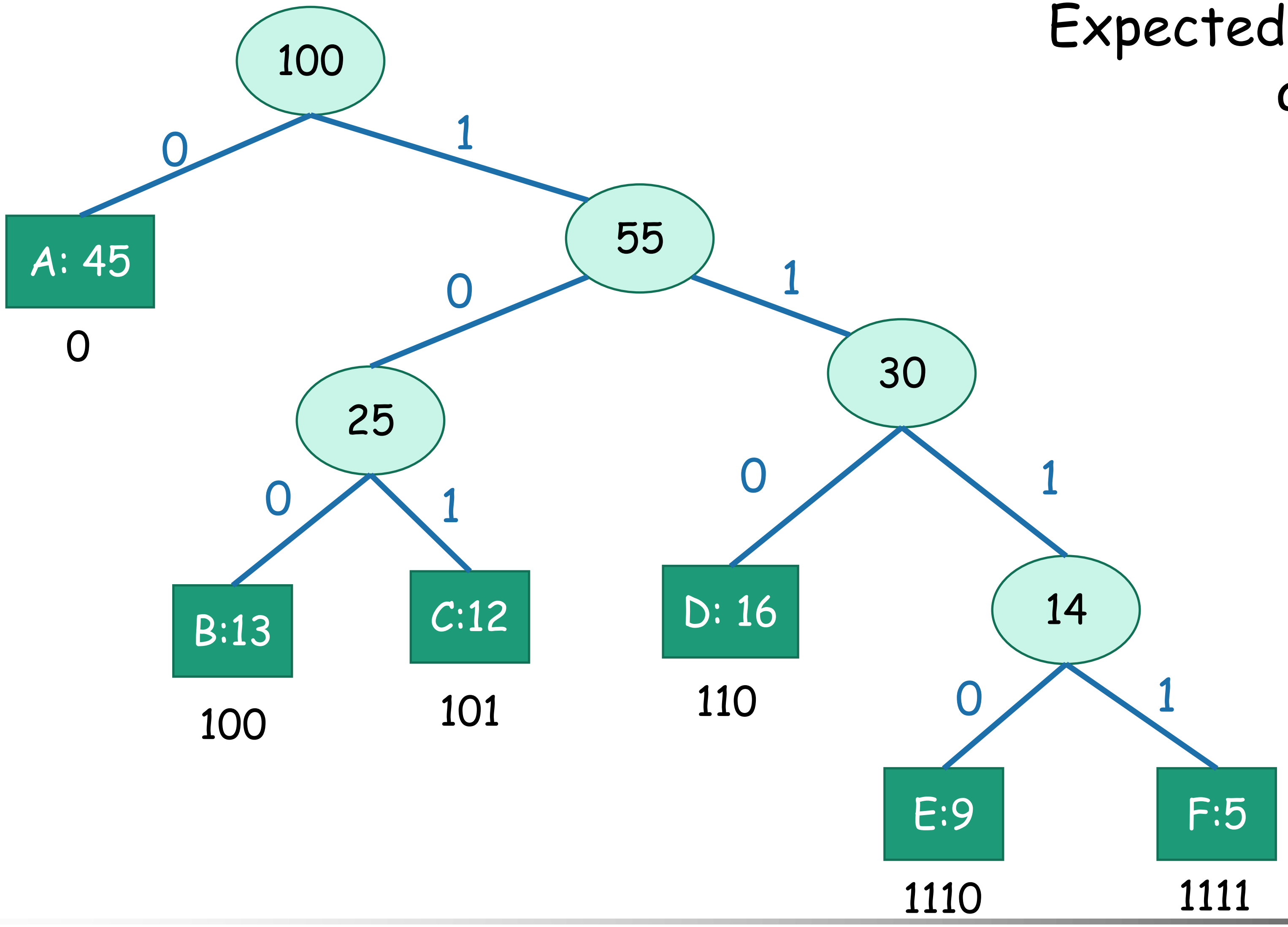
Solution



Solution



Solution

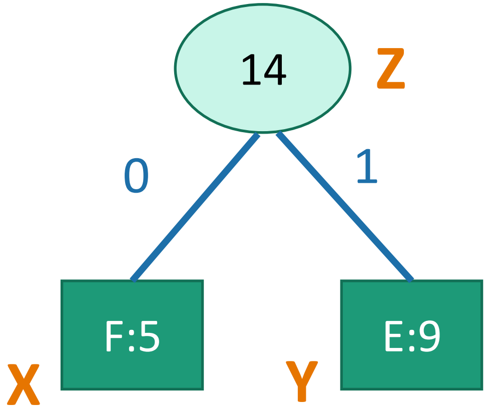


Expected cost of encoding a letter:

$$\mathbf{1 \cdot 0.45}$$
$$+$$
$$\mathbf{3 \cdot 0.41}$$
$$+$$
$$\mathbf{4 \cdot 0.14}$$
$$= \mathbf{2.24}$$

Huffman coding

- Create a node like D: 16 for each letter/frequency
  - The key is the frequency (16 in this case)
- Let CURRENT be the list of all these nodes.
- **while** len(CURRENT) > 1:
  - X and Y ← the nodes in CURRENT with the smallest keys.
  - Create a new node Z with Z.key = X.key + Y.key
  - Set Z.left = X, Z.right = Y
  - Add Z to CURRENT and remove X and Y
- return CURRENT[0]



Does Greedy algorithm always return the best solution?

The 0-1 Knapsack Problem

- Given: A set of n items, with each item i having
  - w_i – a positive weight
  - v_i – a positive benefit value
- Goal: Choose items with maximum total value but with weight at most W.

Example 1

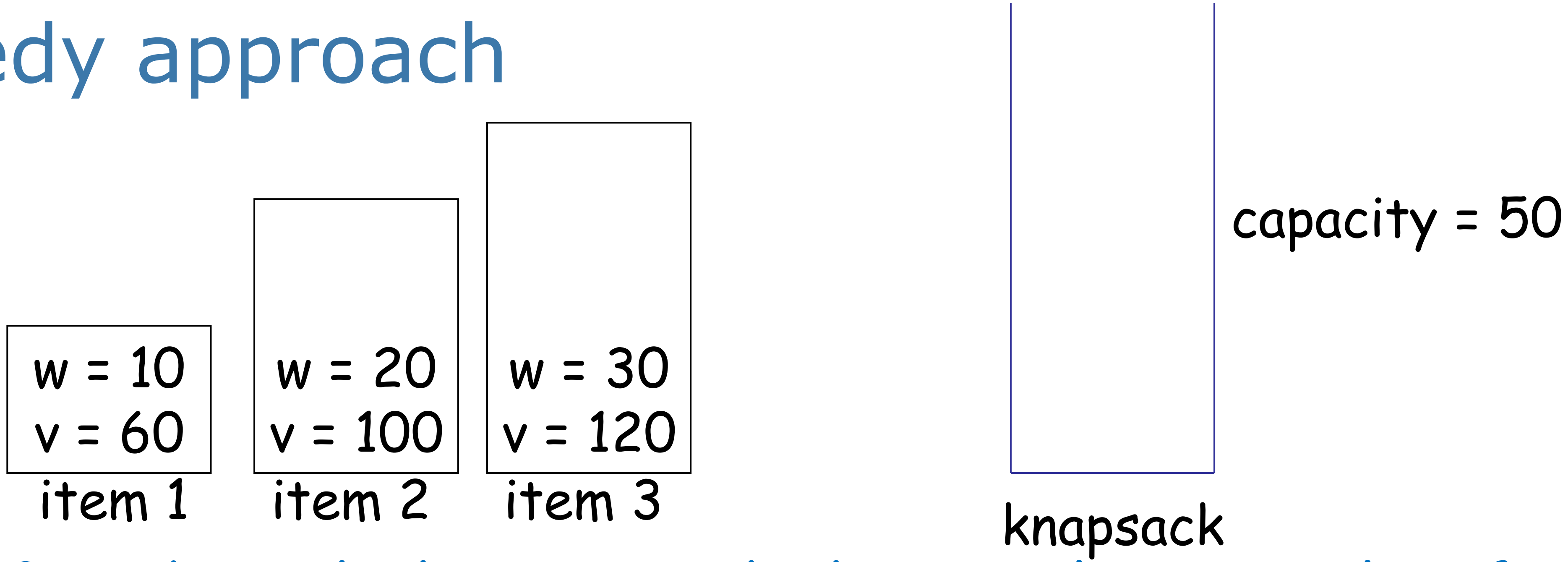
item 1	item 2	item 3
w = 10	w = 20	w = 30
v = 60	v = 100	v = 120

capacity = 50

knapsack

subset	total weight	total value
ϕ	0	0
{1}	10	60
{2}	20	100
{3}	30	120
{1,2}	30	160
{1,3}	40	180
{2,3}	**50**	**220**
{1,2,3}	60	N/A

Greedy approach



Greedy: pick the item with the next largest value if total weight ≤ capacity.

Result:

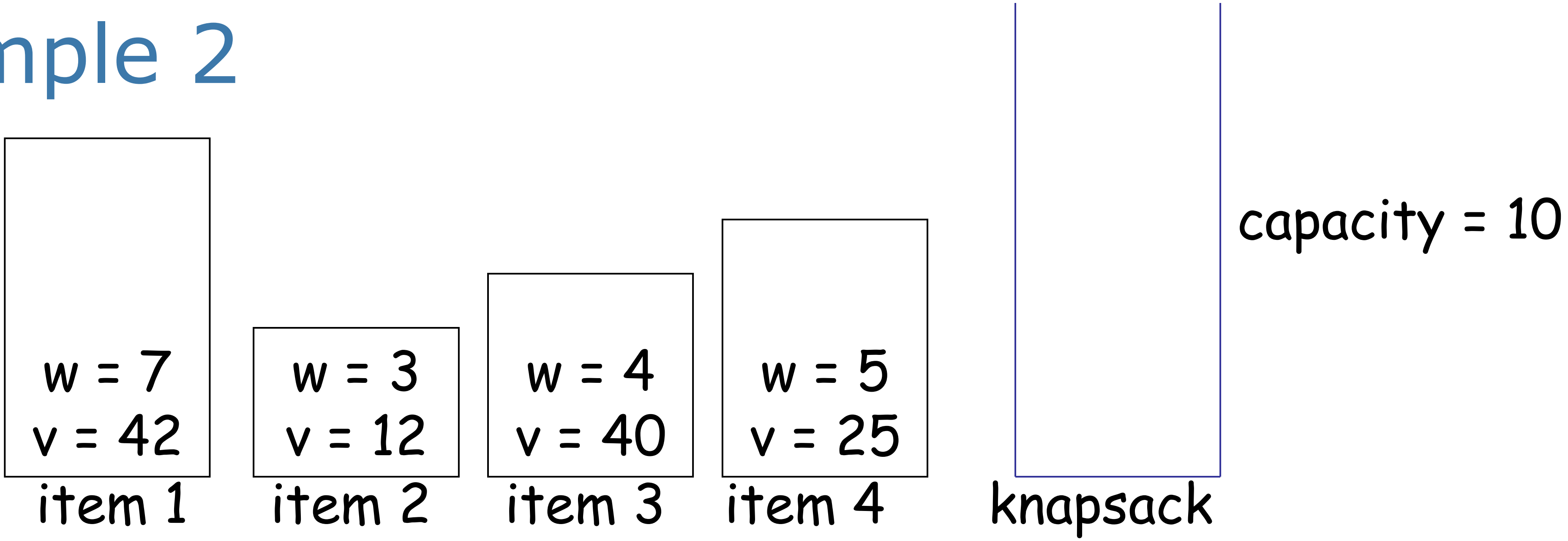
Time complexity?

O(n log n)

- item 3 is taken, total value = 120, total weight = 30
- item 2 is taken, total value = 220, total weight = 50
- item 1 cannot be taken

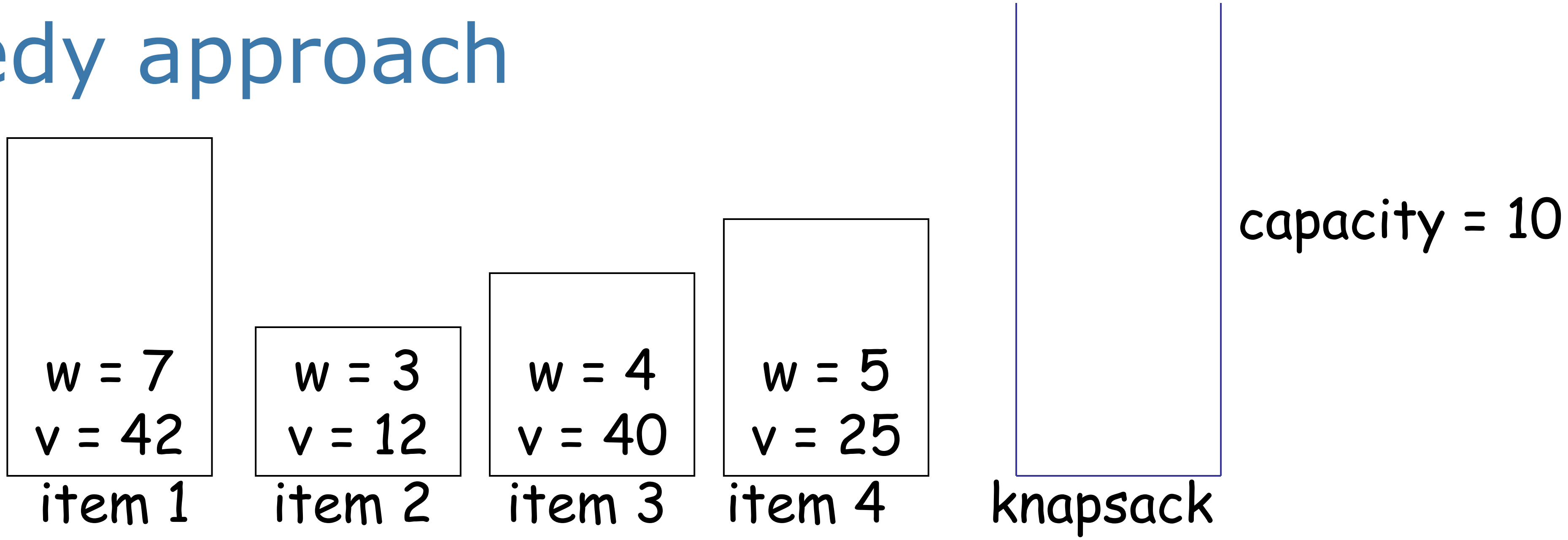
Does this always work?

Example 2



subset	total weight	total value	subset	total weight	total value
ϕ	0	0	{2,3}	7	52
{1}	7	42	{2,4}	8	37
{2}	3	12	**{3,4}**	**9**	**65**
{3}	4	40	{1,2,3}	14	N/A
{4}	5	25	{1,2,4}	15	N/A
{1,2}	10	54	{1,3,4}	16	N/A
{1,3}	11	N/A	{2,3,4}	12	N/A
{1,4}	12	N/A	{1,2,3,4}	19	N/A

Greedy approach

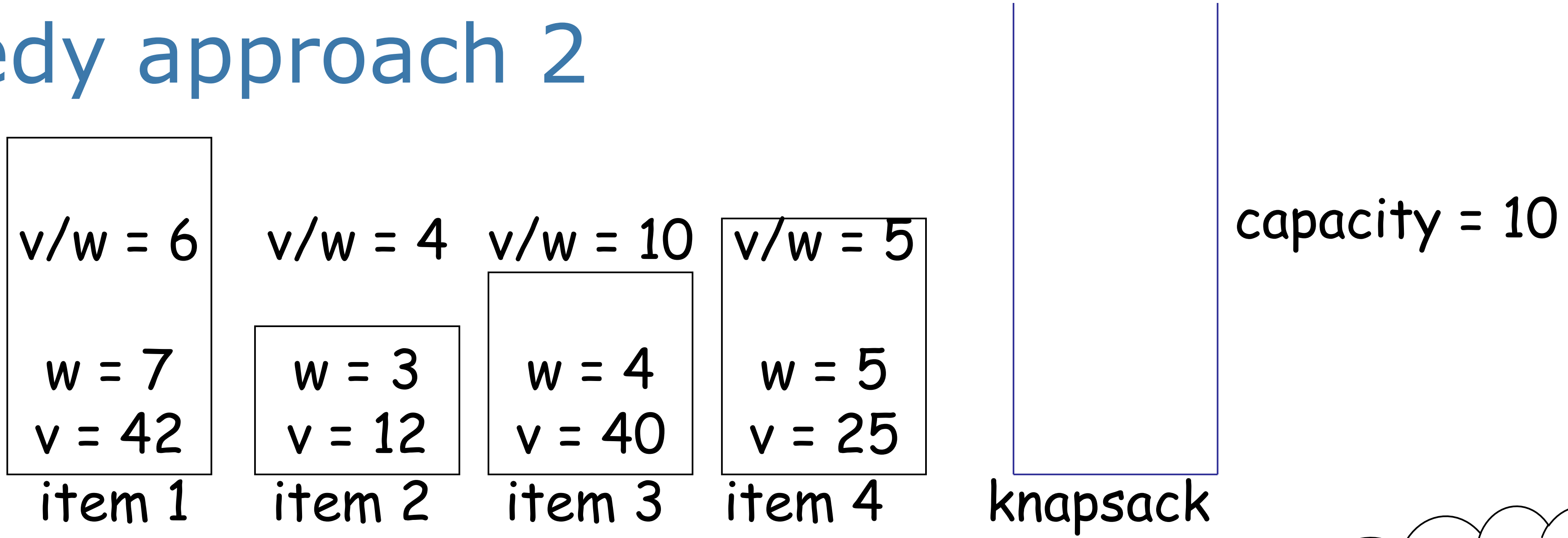


Greedy: pick the item with the next largest value if total weight ≤ capacity.

Result:

- item 1 is taken, total value = 42, total weight = 7
- item 3 cannot be taken
- item 4 cannot be taken
- item 2 is taken, total value = 54, total weight = 10

Greedy approach 2



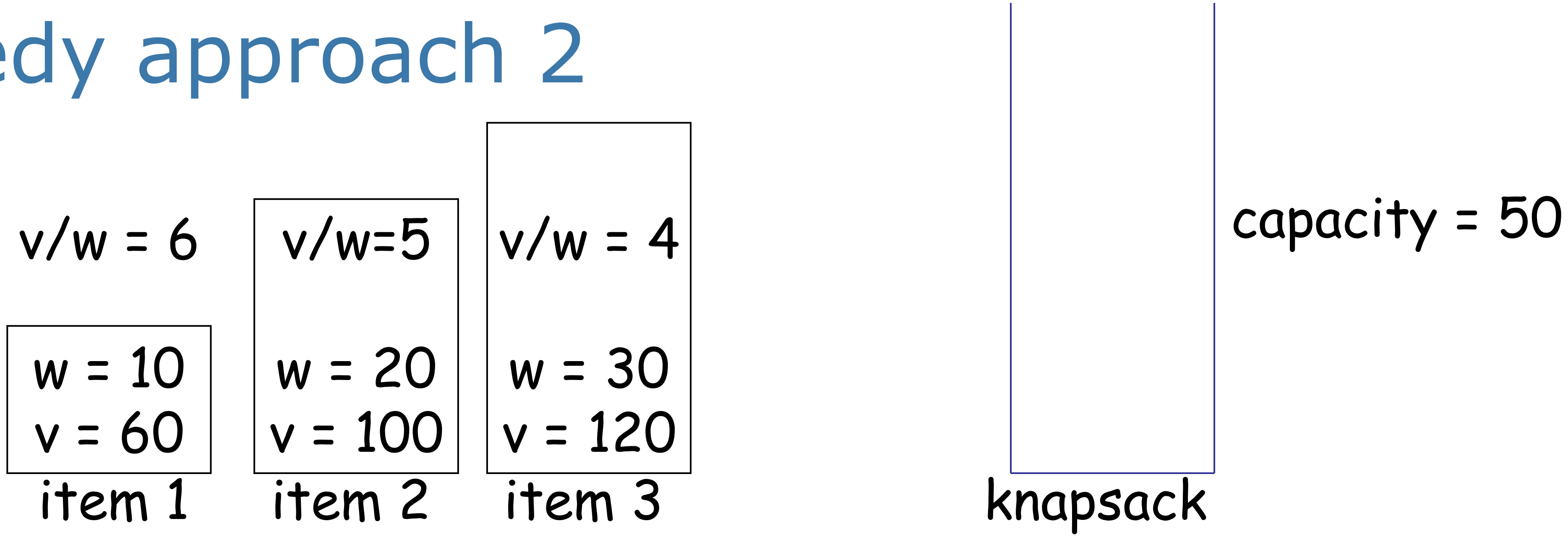
Greedy 2: pick the item with the next largest (value/weight) if total weight ≤ capacity.

Work for Eg 1?

Result:

- item 3 is taken, total value = 40, total weight = 4
- item 1 cannot be taken
- item 4 is taken, total value = 65, total weight = 9
- item 2 cannot be taken

Greedy approach 2



Greedy: pick the item with the next largest (value/weight) if total weight ≤ capacity.

Result:

- item 1 is taken, total value = 60, total weight = 10
- item 2 is taken, total value = 160, total weight = 30
- item 3 cannot be taken

Not the best!!

Greedy Algorithms

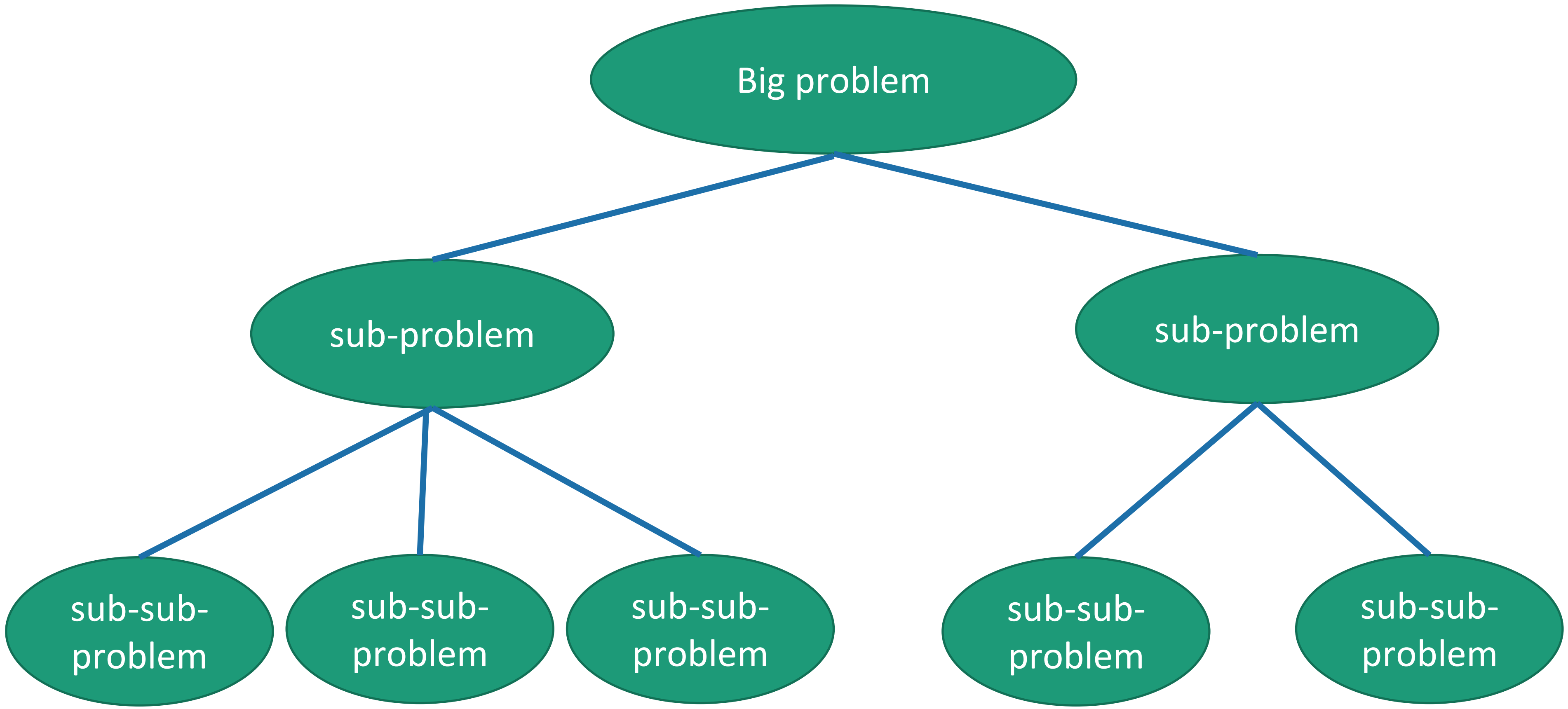
Advantages

- Don't need to pay much effort at each step
- Usually finds a solution very **quickly**
- The solution found is usually **not bad**

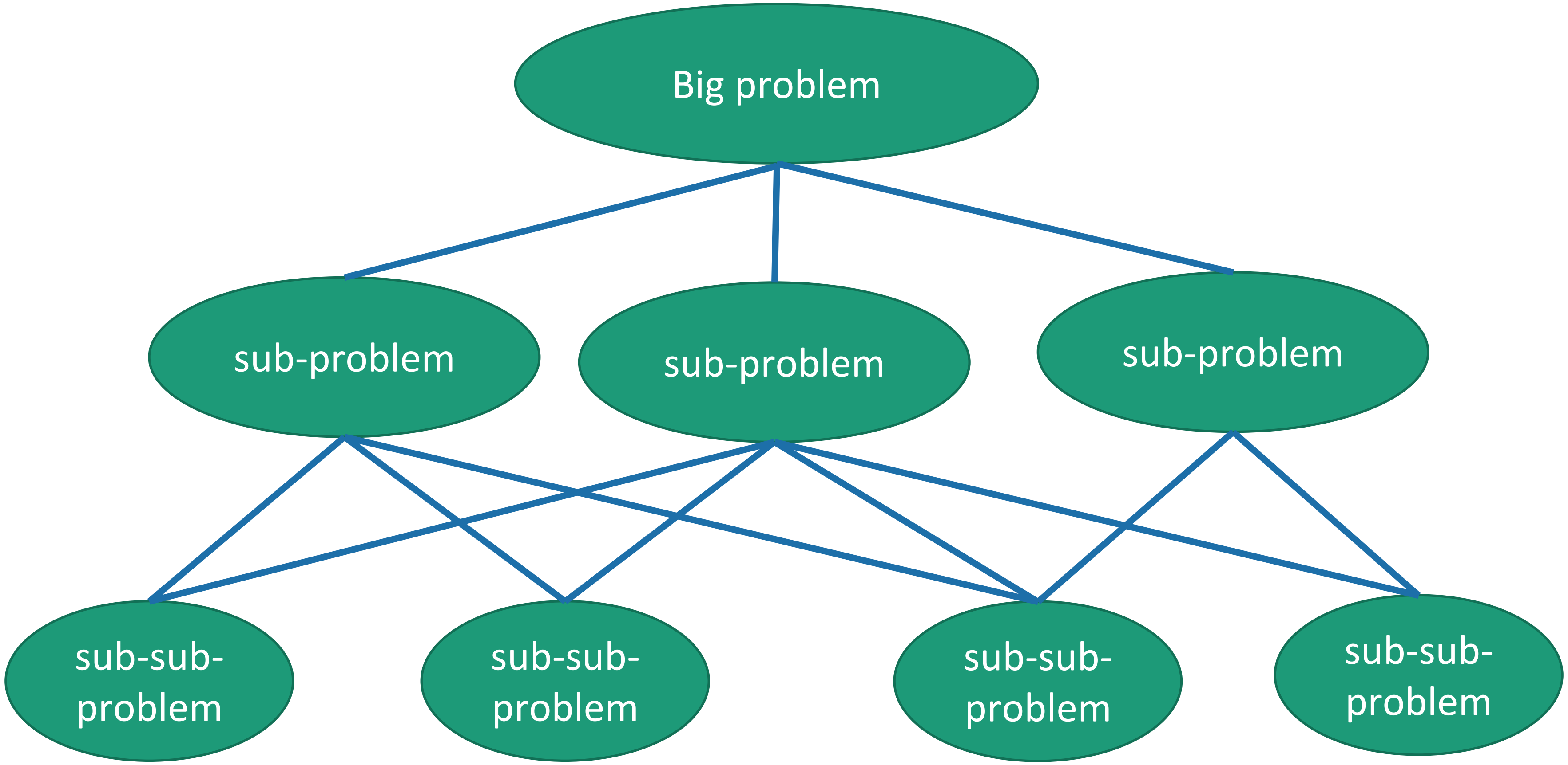
Possible problem

- The solution found may **NOT** be the best one

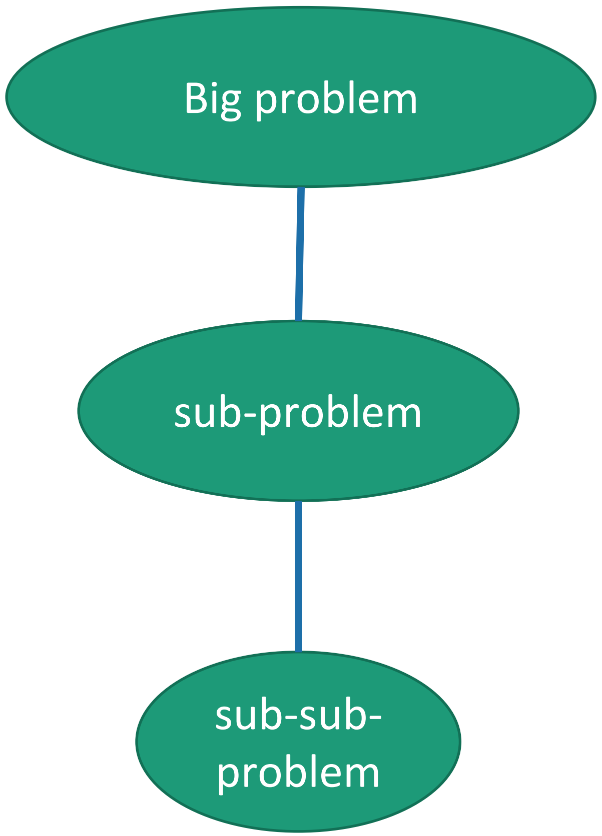
Divide-and-conquer



Dynamic Programming



Greedy algorithms



Optimal solutions to a problem are made up from optimal solutions of sub-problems

Each problem **depends on only one sub-problem.**

Optional Exercise

- Given *2n* integers
- Group these integers into n pairs (a_1,b_1), (a_2,b_2),…, (a_n,b_n)
- Find the maximized sum of min (a_i,b_i) for all i.

- For example: [1,4,2,3]
  - min(1,4) + min(2,3) = 3
  - min(1,2) + min(3,4) has the maximum sum 4

Learning outcome

- Understand what greedy algorithm is
- Able to apply greedy algorithm to solve
  - the Activity Selection problem
  - the Huffman Coding problem
- Able to apply greedy algorithm to find solution for Knapsack problem