# Giải thuật tham lam

Greedy algorithms

# Giải thuật tham lam (Greedy algorithms)

- Mỗi thời điểm chỉ chọn một cách
- Không quay lui
- Hi vọng lựa chọn là hợp lý nhất

Sử dụng một phần tài liệu bài giảng CS161 Stanford University

# Nội dung

- Phản ví dụ:
  - Knapsack
- 3 ví dụ greedy algorithms:
  - Activity Selection (lựa chọn hành động)
  - Job Scheduling (lập lịch)
  - Mã hóa Huffman

Capacity: 10

| Item: |  |  |  |  |  |
|---|---|---|---|---|---|
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

- Unbounded Knapsack:
  - Suppose I have infinite copies of all items.
  - What's the most valuable way to fill the knapsack?

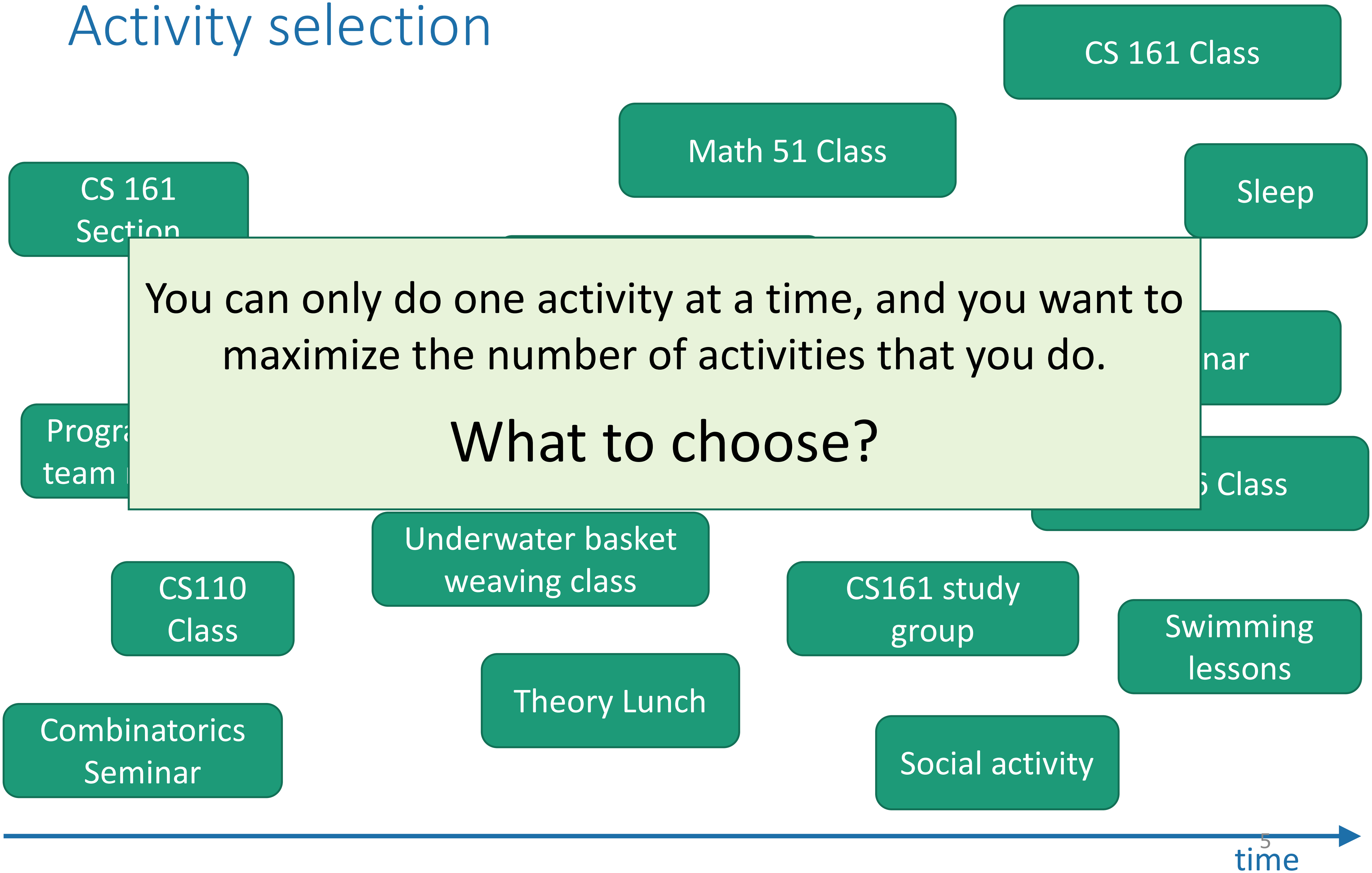
Total weight: 10
Total value: 42

- "Greedy" algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

Total weight: 9
Total value: 39

# Example where greedy works

## Activity selection

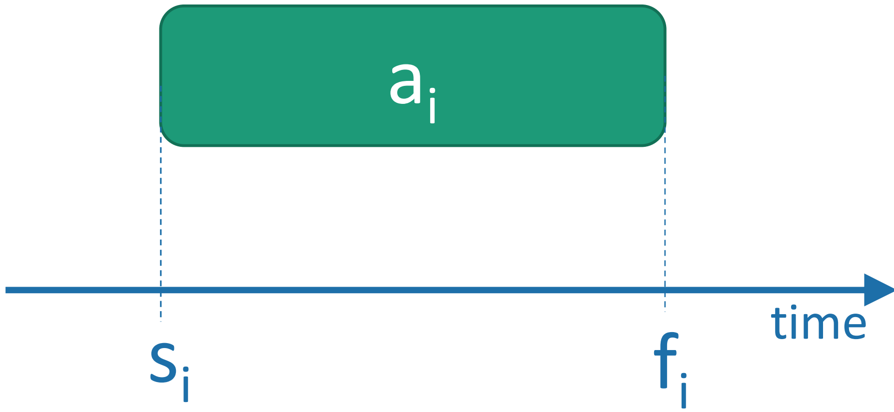



You can only do one activity at a time, and you want to maximize the number of activities that you do.

**What to choose?**

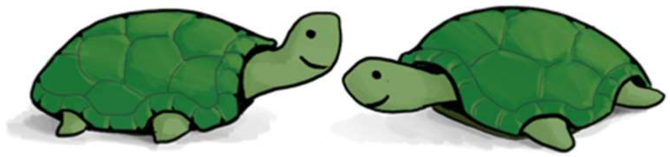
# Activity selection

- Input:
  - Activities $a_1$, $a_2$, …, $a_n$
  - Start times $s_1$, $s_2$, …, $s_n$
  - Finish times $f_1$, $f_2$, …, $f_n$



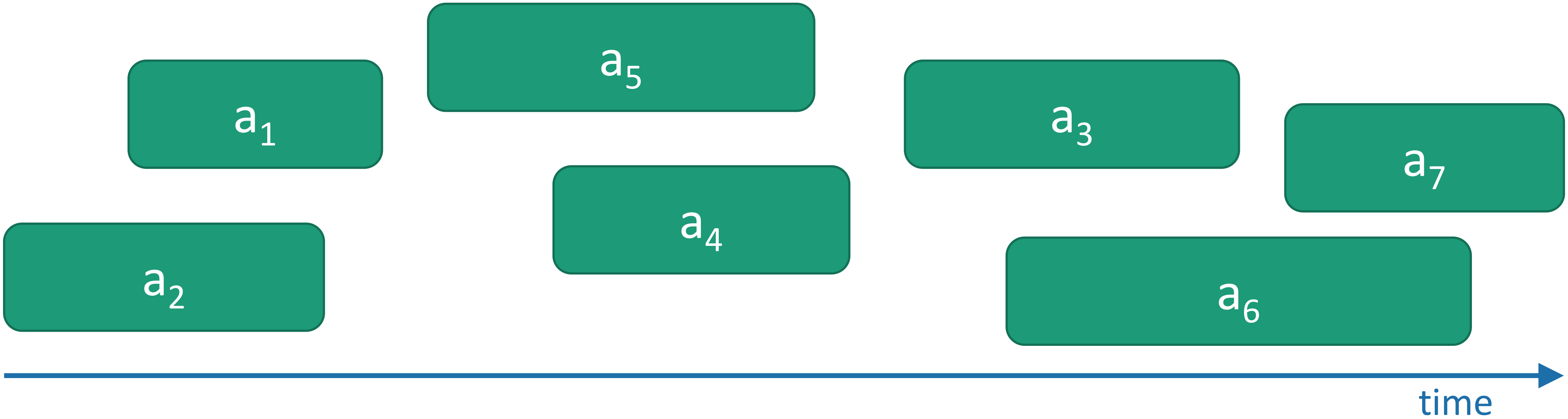

- Output:
  - A way to maximize the number of activities you can do today.

In what order should you greedily add activities?



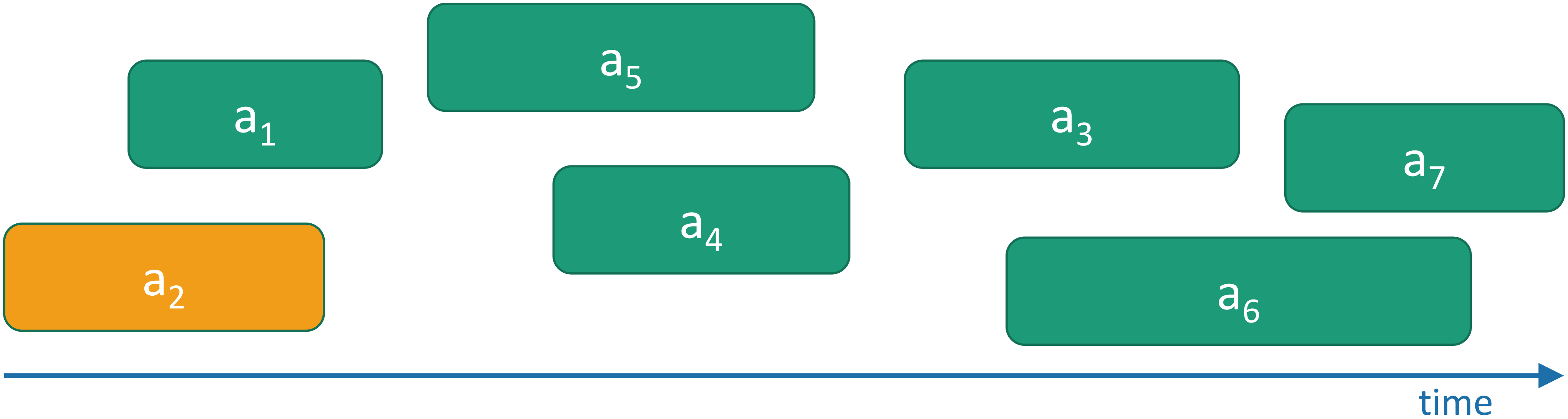
Think-share!
1 minute think; (wait) 1 minute share

# Greedy Algorithm



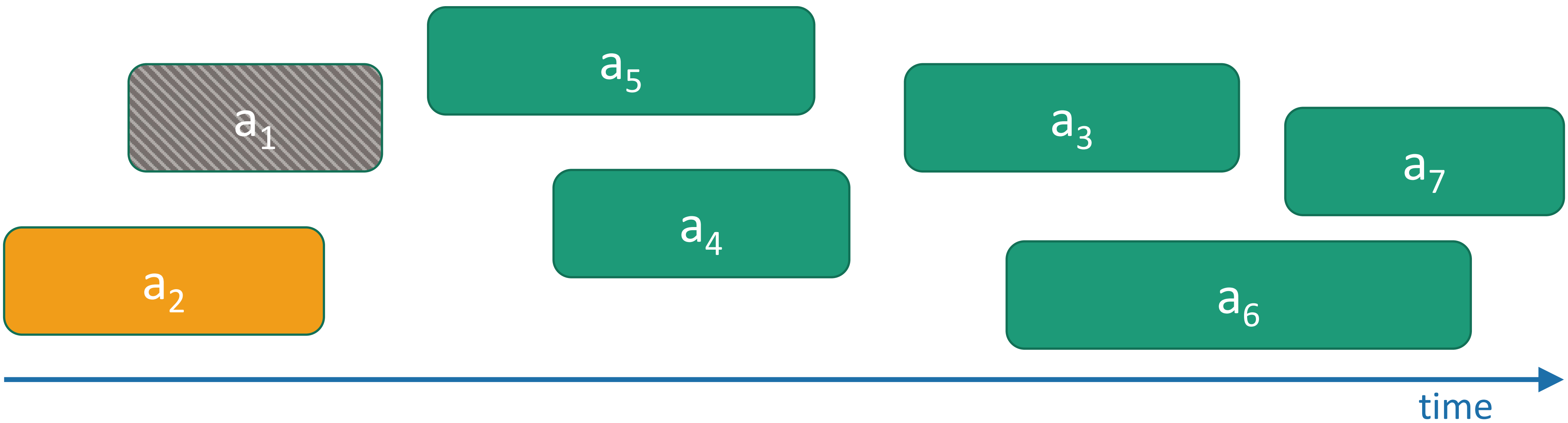

- Pick activity you can add with the smallest finish time.
- Repeat.

# Greedy Algorithm



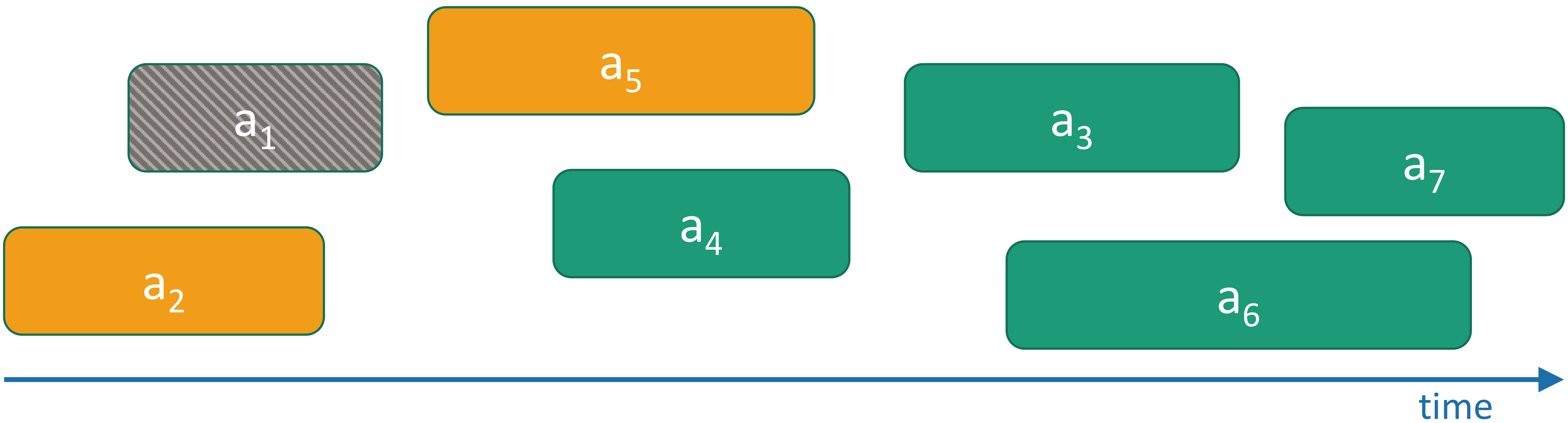

- Pick activity you can add with the smallest finish time.
- Repeat.

# Greedy Algorithm



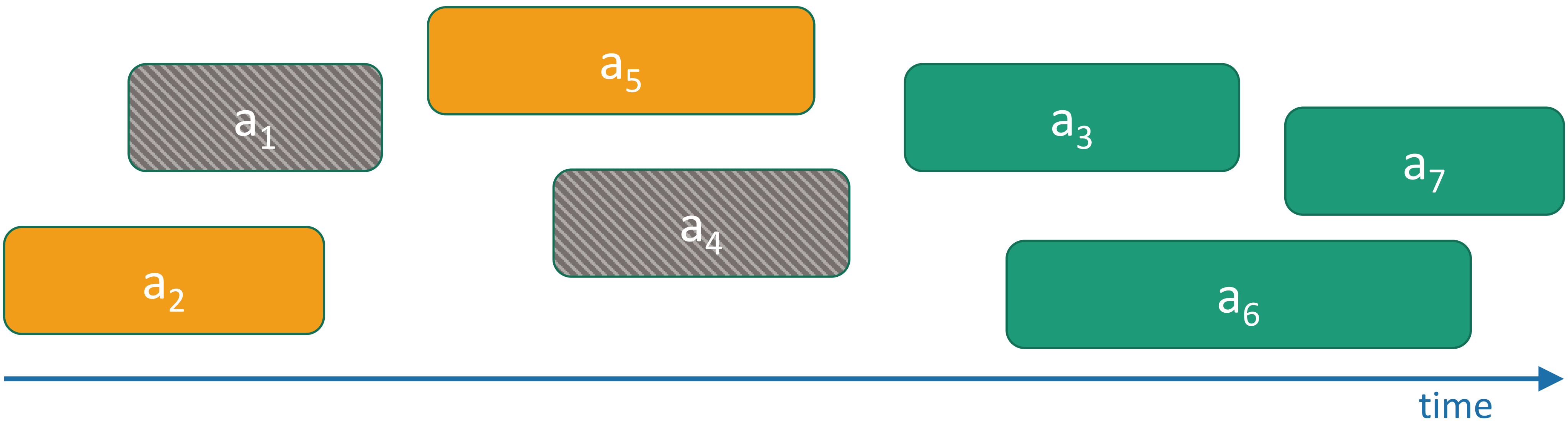

- Pick activity you can add with the smallest finish time.
- Repeat.

# Greedy Algorithm



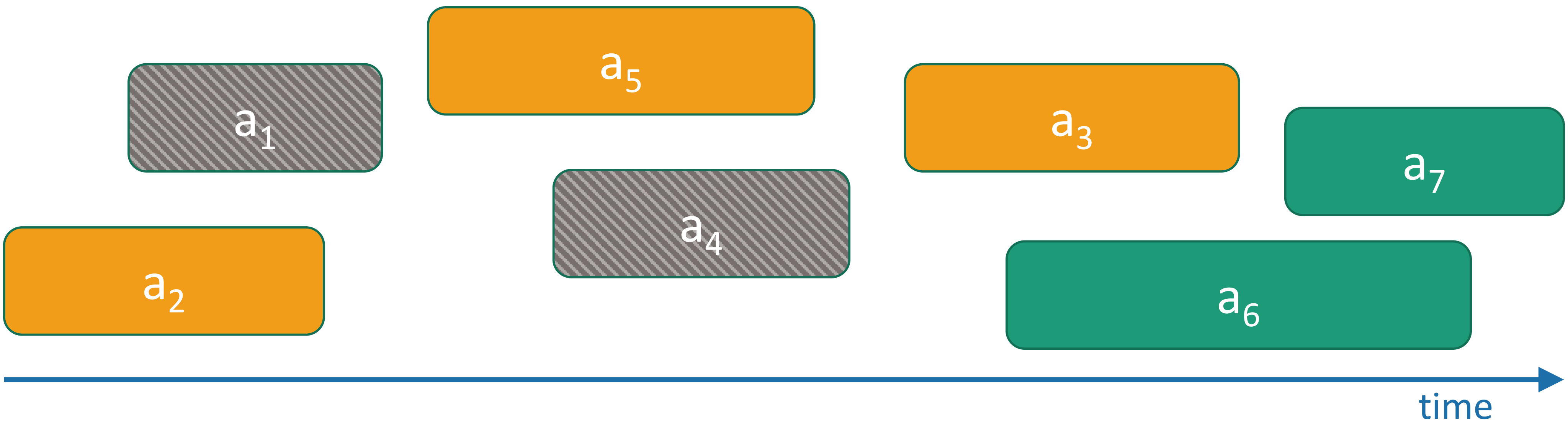

- Pick activity you can add with the smallest finish time.
- Repeat.

# Greedy Algorithm



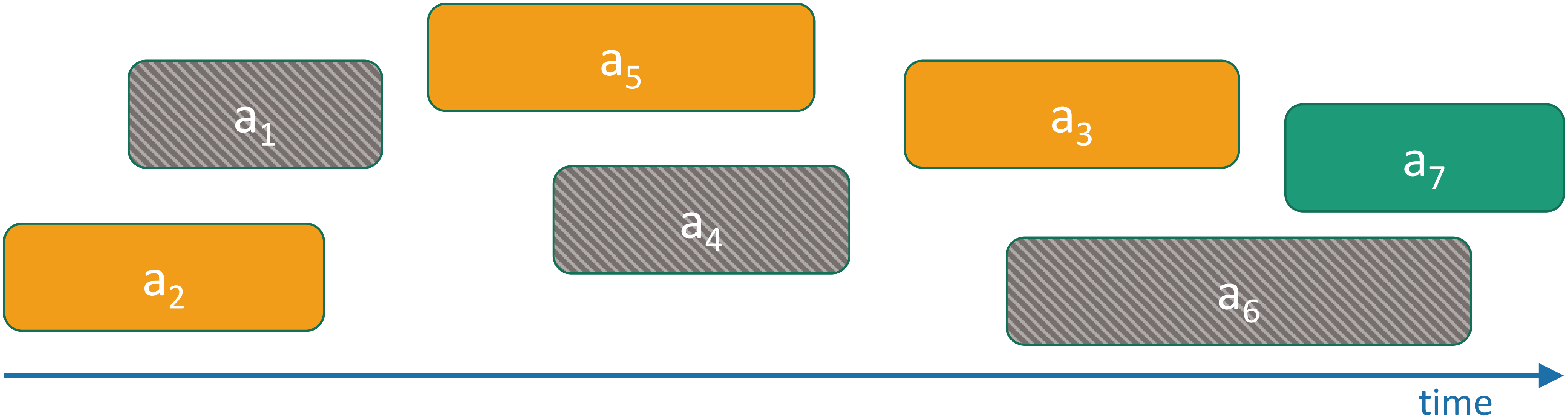

- Pick activity you can add with the smallest finish time.
- Repeat.

# Greedy Algorithm



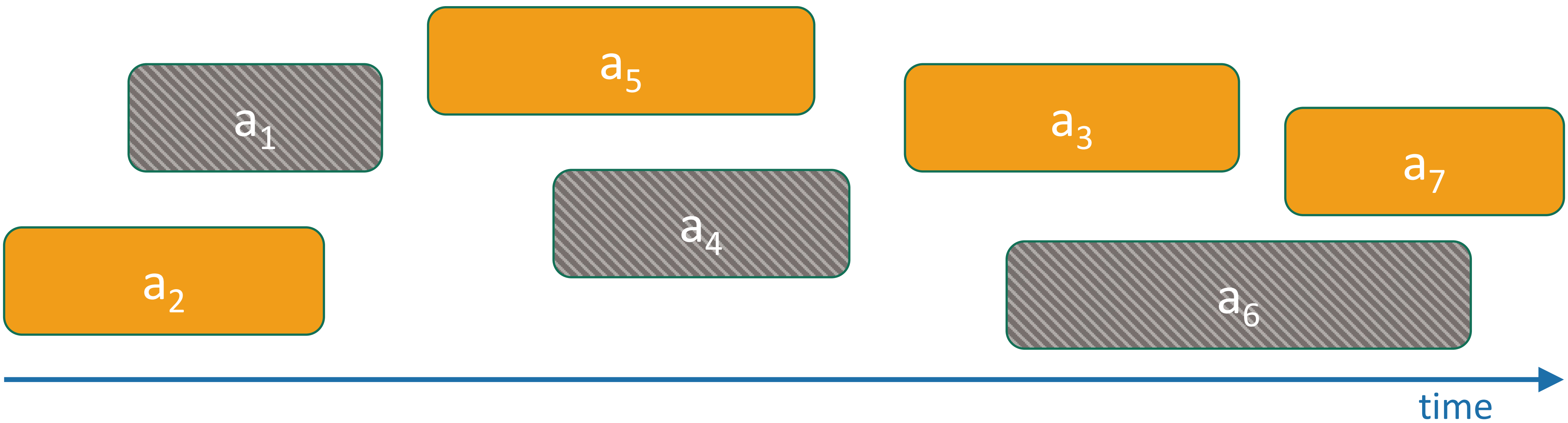

- Pick activity you can add with the smallest finish time.
- Repeat.

# 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.
  - Hey, I can increase my activity set by one,
  - And leave lots of room for future choices,
  - Let's do that and hope for the best!!!

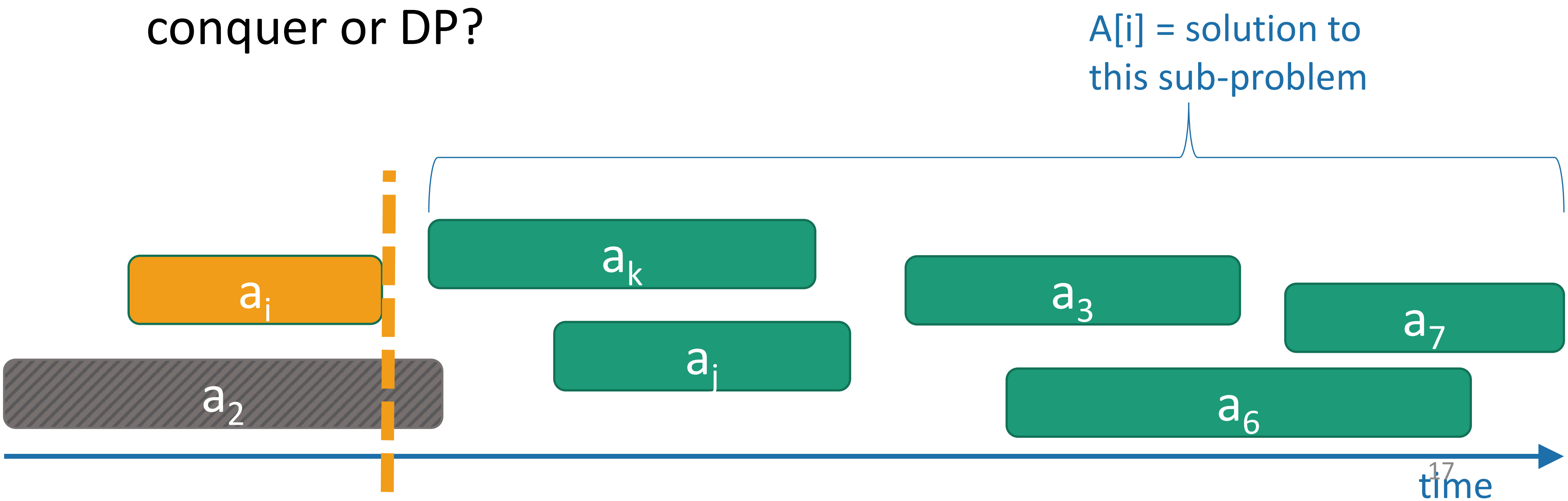
- **Hope** that at the end of the day, this results in a globally optimal solution.

# Optimal sub-structure
## in greedy algorithms

- Our greedy activity selection algorithm exploited a natural sub-problem structure:

  A[i] = number of activities you can do after the end of activity i

- How does this substructure relate to that of divide-and-conquer or DP?



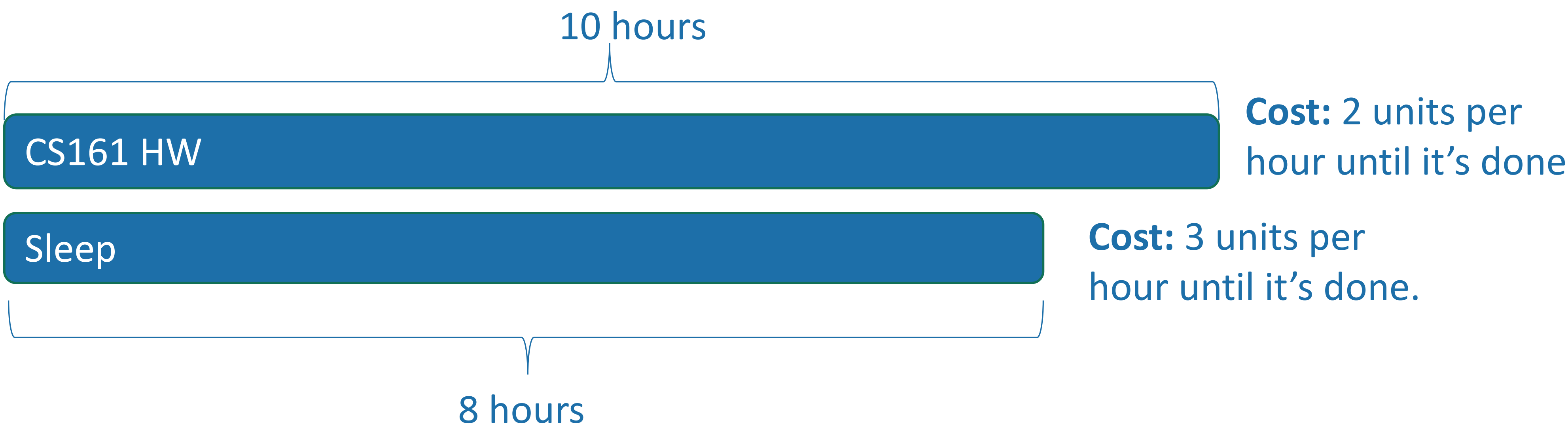
Let’s see a few more examples

# Another example:
# Scheduling

- CS161 HW
- Personal hygiene
- Math HW
- Administrative stuff for student club
- Econ HW
- Do laundry
- Meditate
- Practice musical instrument
- Read lecture notes
- Have a social life
- Sleep

# Scheduling

- n tasks
- Task i takes $t_i$ hours
- For every hour that passes until task i is done, pay $c_i$




- CS161 HW, then Sleep: costs **10 · 2 + (10 + 8) · 3 = 74 units**
- Sleep, then CS161 HW: costs **8 · 3 + (10 + 8) · 2 = 60 units**

# Optimal substructure

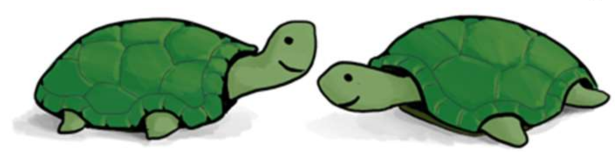
- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:

Job A | Job B | Job C | Job D

**Then this must be the optimal schedule on just jobs B,C,D.**

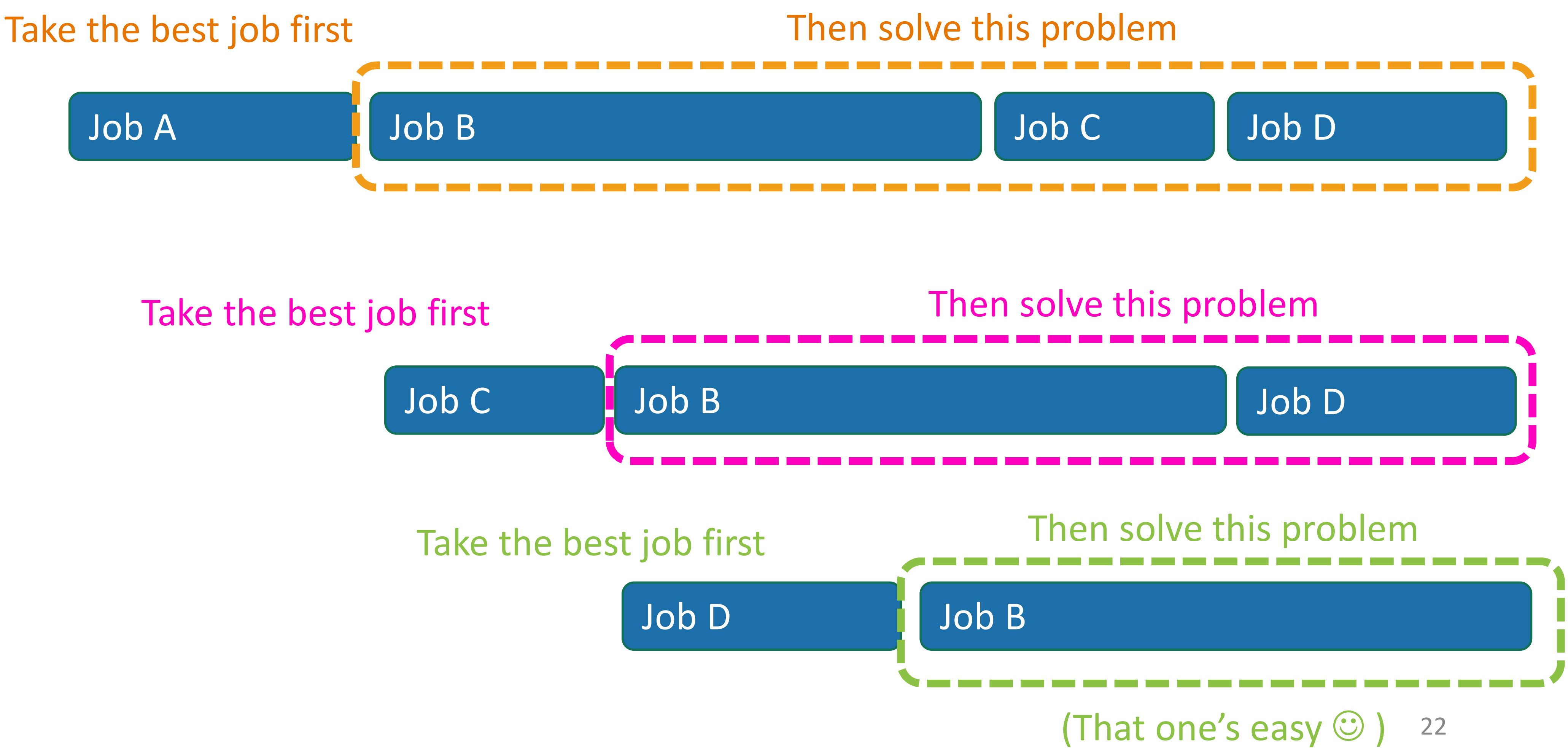
Why?



Think-share
1 minute think
(wait) 1 minute share

# Optimal substructure

- Seems amenable to a greedy algorithm:



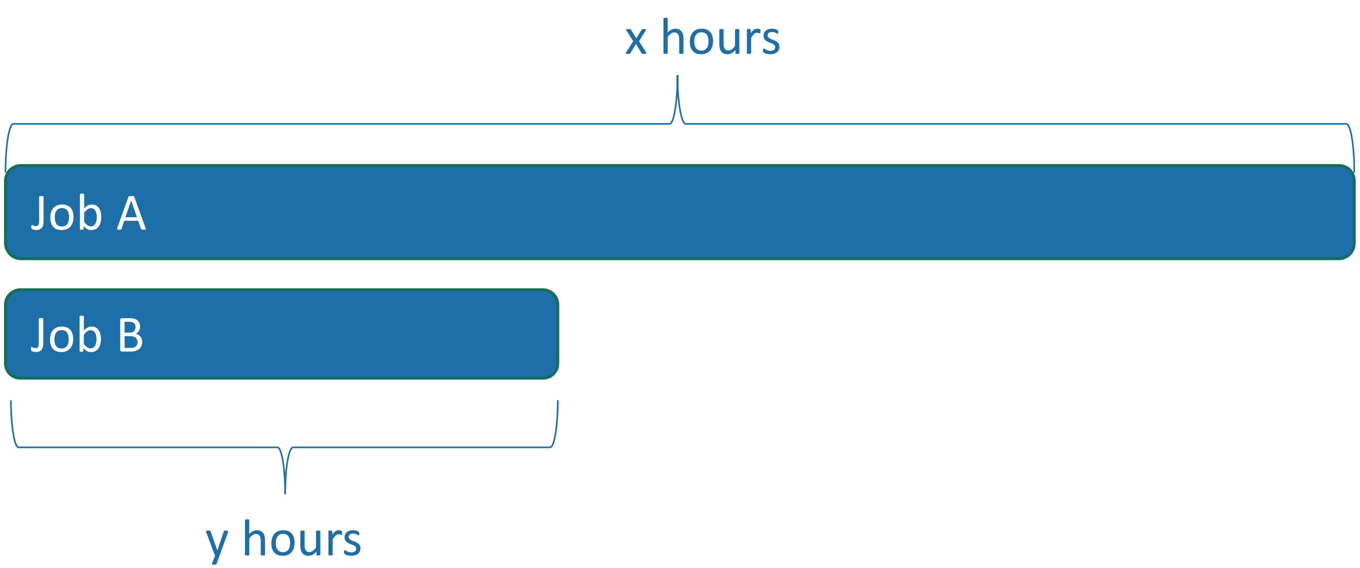
# What does “best” mean?

Note: here we are defining x, y, z, and w. (We use $c_i$ and $t_i$ for these in the general problem, but we are changing notation for just this thought experiment to save on subscripts.)

**AB** is better than **BA** when:

$$xz + (x+y)w \leq yw + (x+y)z$$
$$xz + xw + yw \leq yw + xz + yz$$
$$wx \leq yz$$
$$\frac{w}{y} \leq \frac{z}{x}$$

- Of these two jobs, which should we do first?




**Cost: z** units per hour until it’s done.

**Cost: w** units per hour until it’s done.

- Cost( **A then B** ) = x · z + (x + y) · w
- Cost( **B then A** ) = y · w + (x + y) · z

What matters is the ratio:

$$\frac{\text{cost of delay}}{\text{time it takes}}$$

“Best” means biggest ratio.

# Idea for greedy algorithm

- Choose the job with the biggest $\frac{\textbf{cost of delay}}{\textbf{time it takes}}$ ratio.

# Greedy Scheduling Solution

- **scheduleJobs**( JOBS ):
  - Sort JOBS in decreasing order by the ratio:
    - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job i}}{\text{time job i takes to complete}}$
  - **Return** JOBS

Running time: O(nlog(n))

Now you can go about your schedule peacefully, in the optimal way.

# One more example

## 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

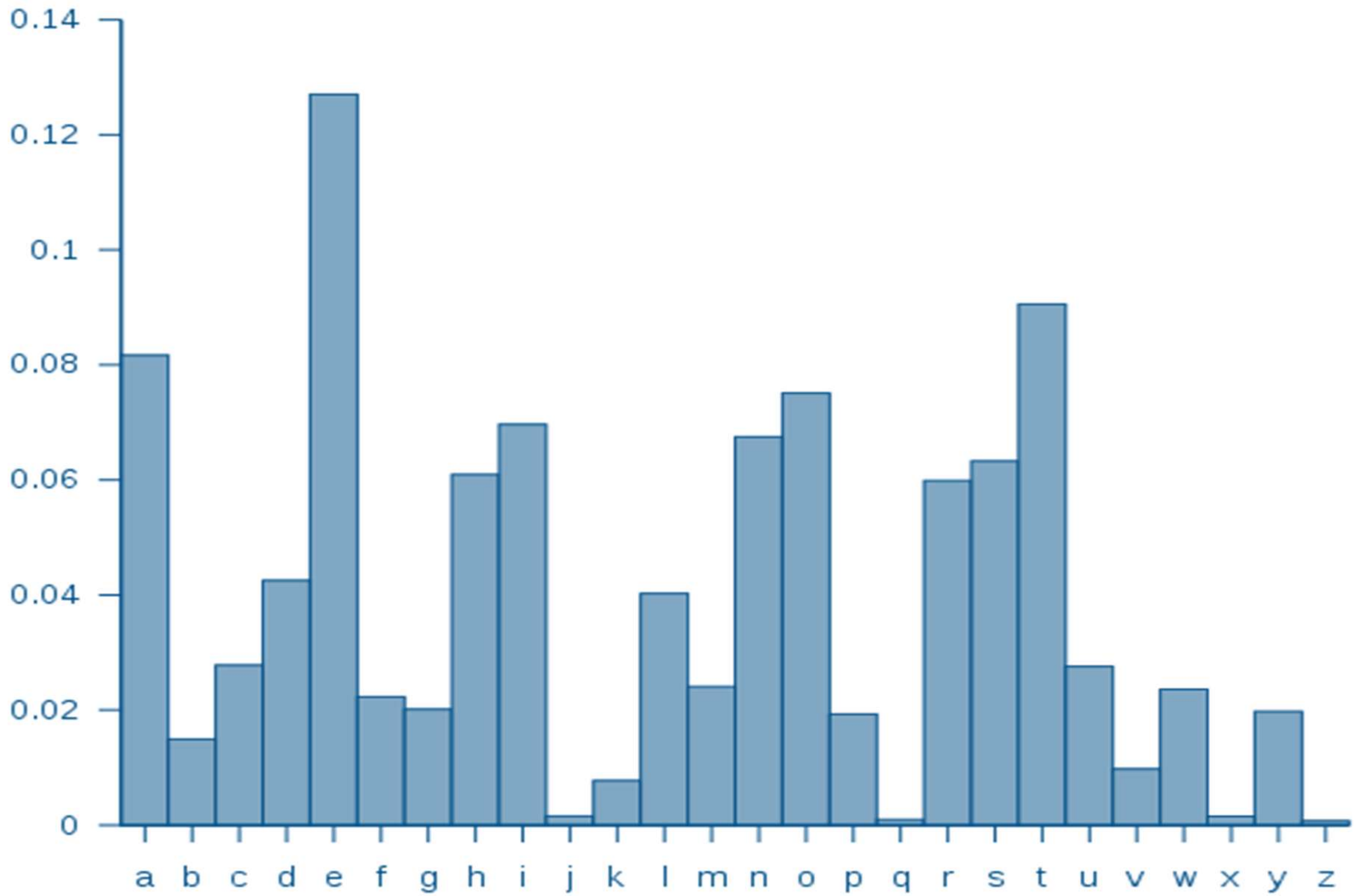
# One more example
## Huffman coding

ASCII is pretty wasteful for English sentences. If **e** shows up so often, we should have a shorter way of representing it!

- **e**v**e**ryday **e**nglish s**e**nt**e**nc**e**
- **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**

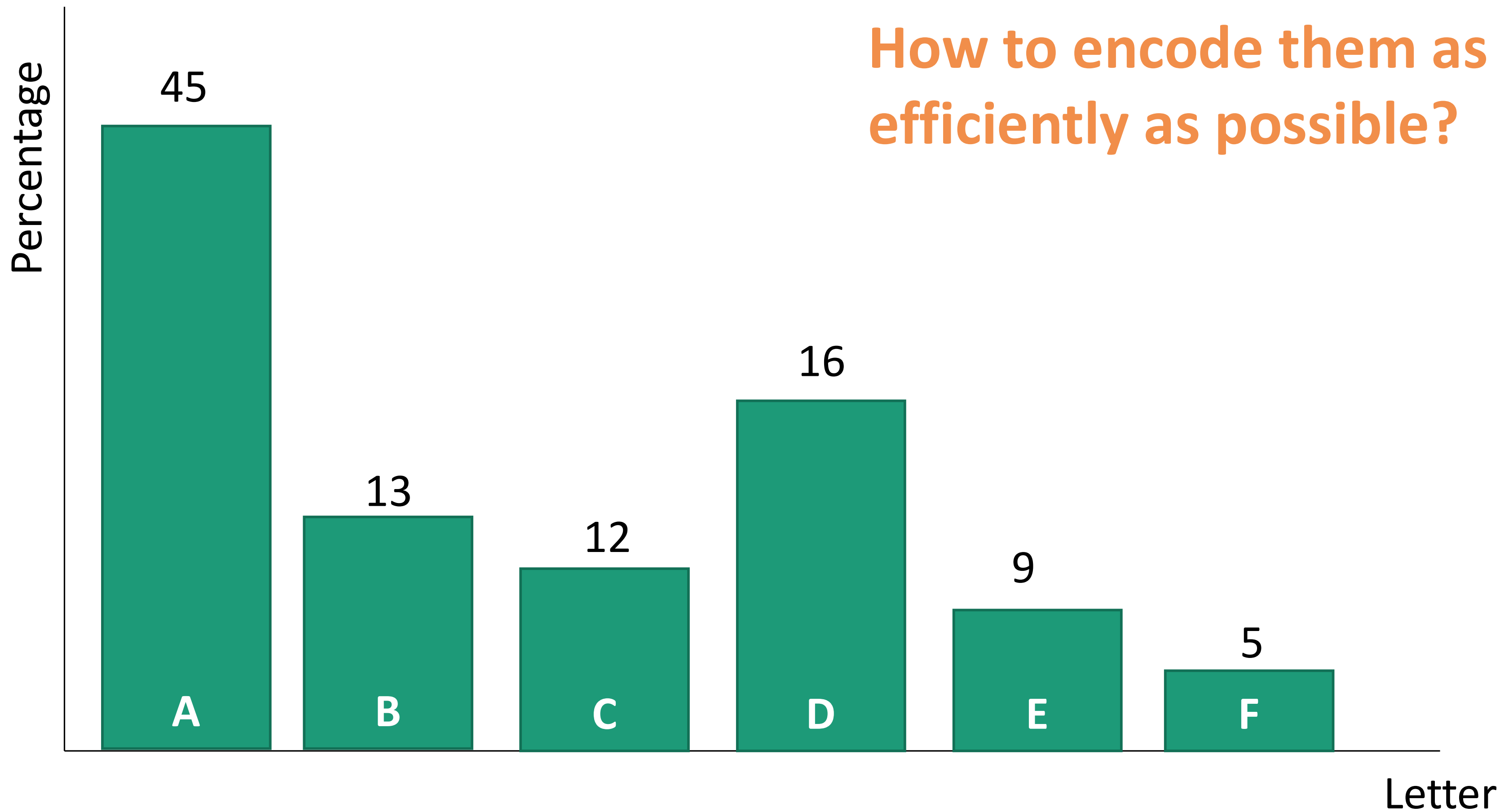
- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

# Suppose we have some distribution on characters



# Suppose we have some distribution on characters

For simplicity, let's go with this made-up example



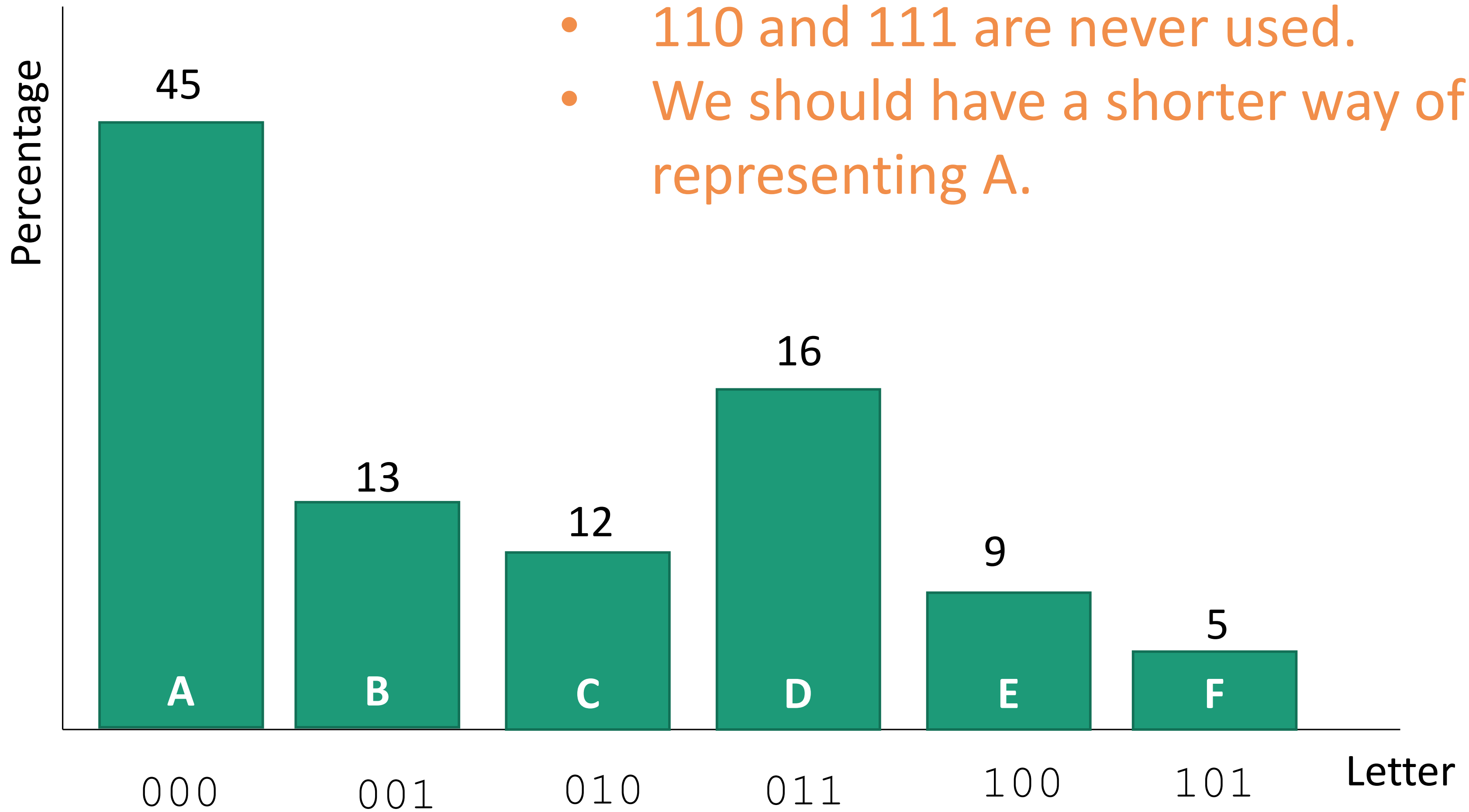

**How to encode them as efficiently as possible?**

# Try 0
(like ASCII)

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

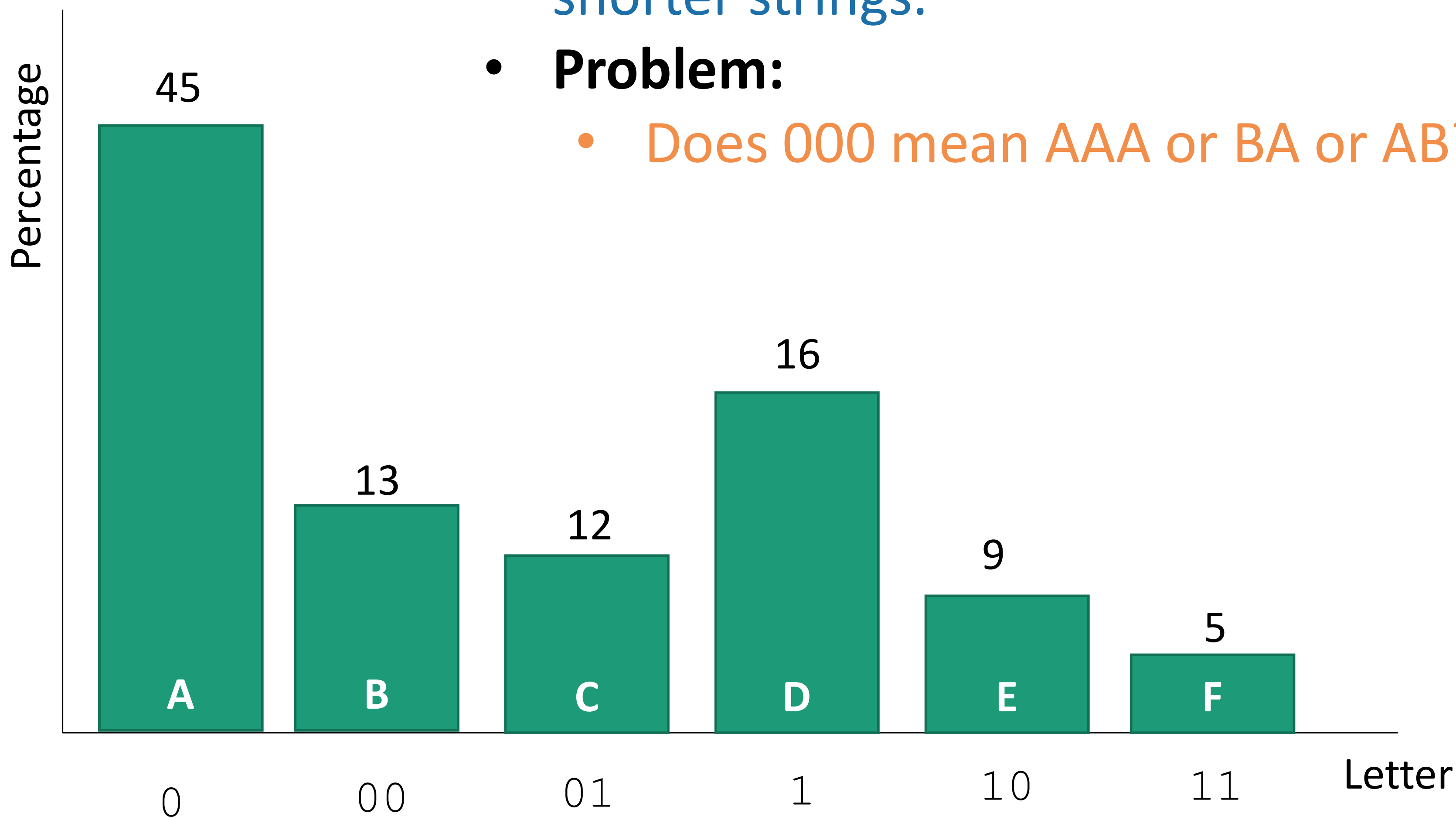
**Wasteful!**

- 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 or BA or AB?

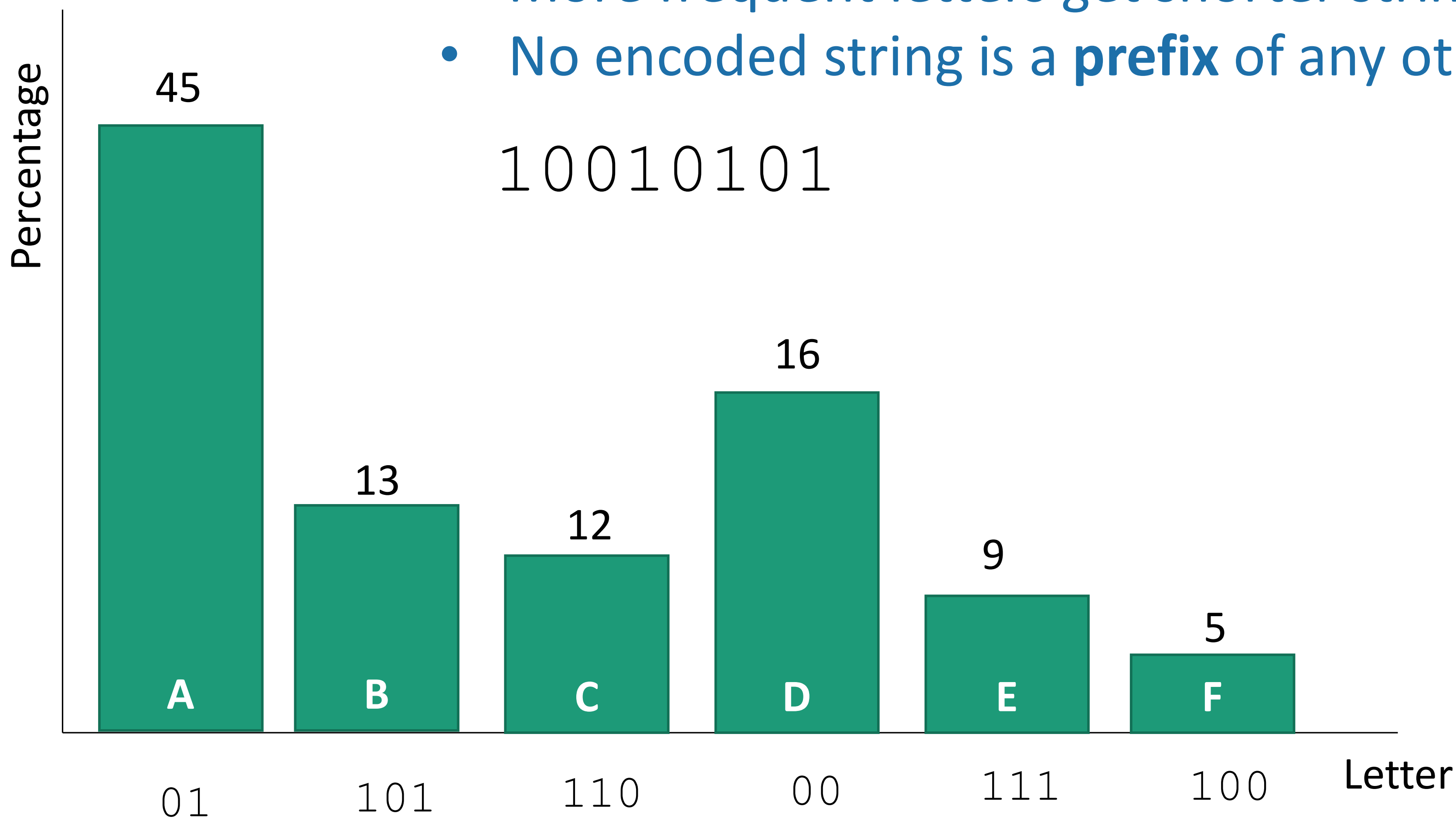


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (e.g. in CLRS).

# Try 2: prefix-free coding

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

10010101

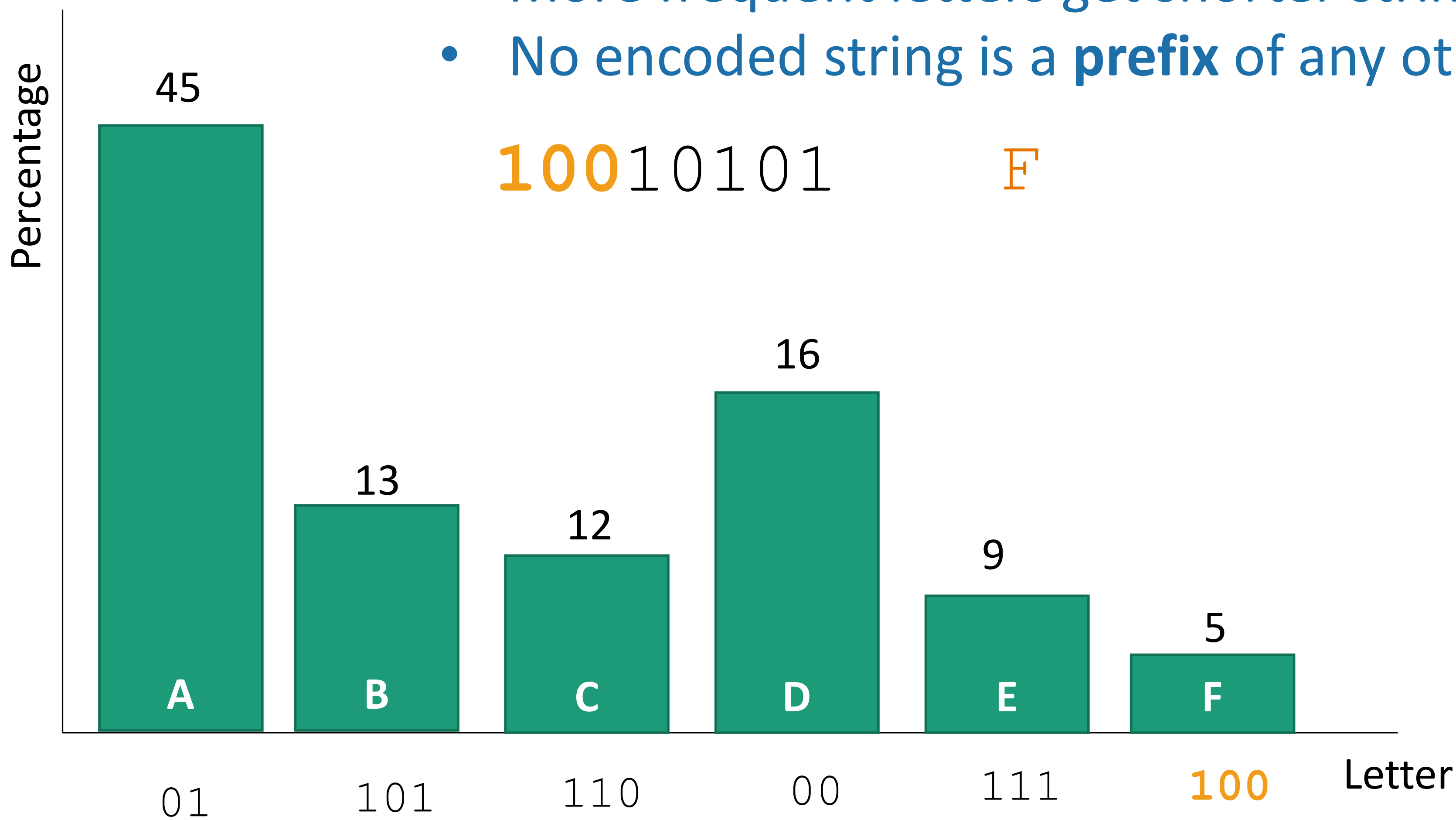


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

# Try 2: prefix-free coding

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

100 10101 F

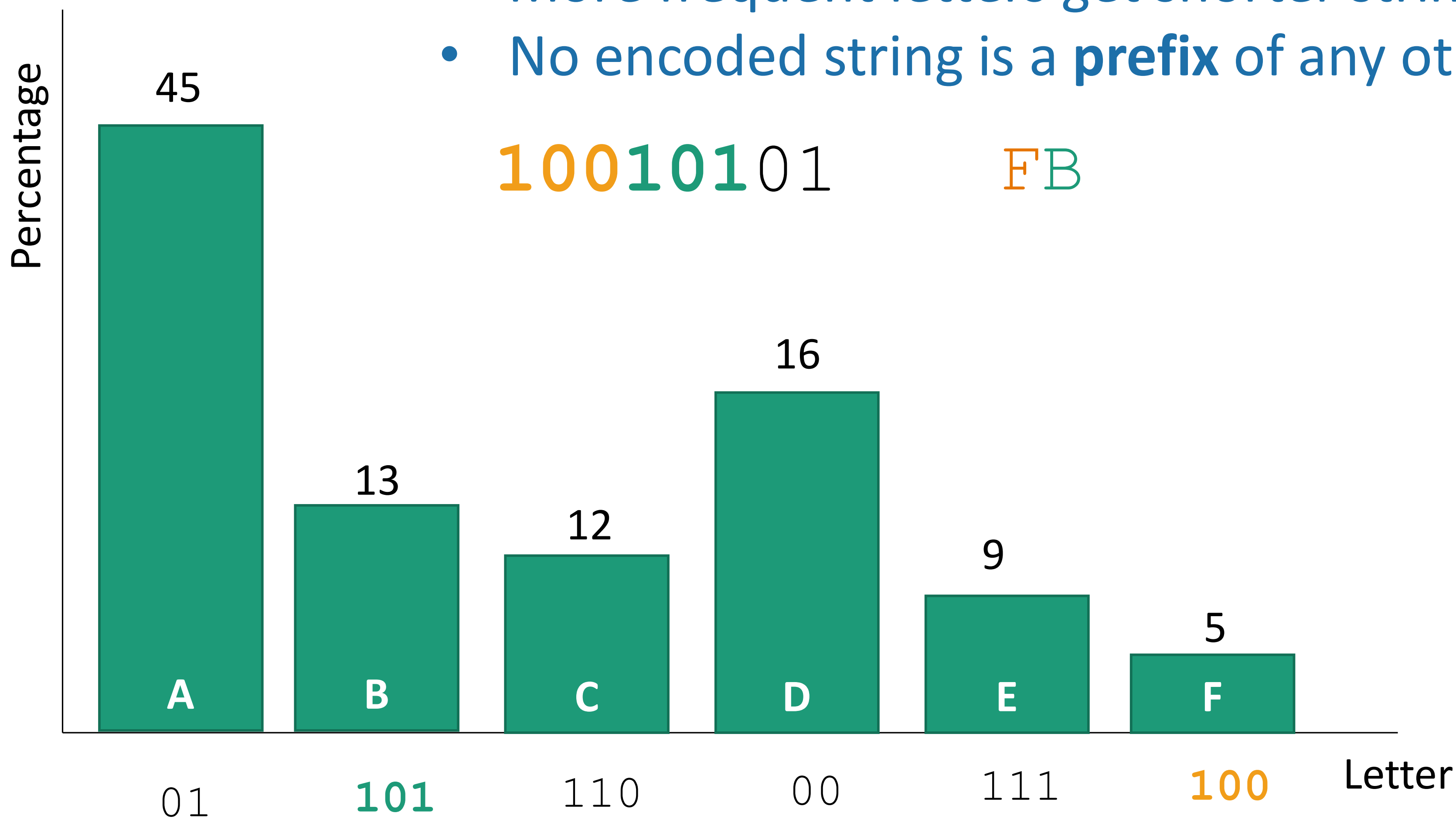


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

# Try 2: prefix-free coding

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

10010101 FB



Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

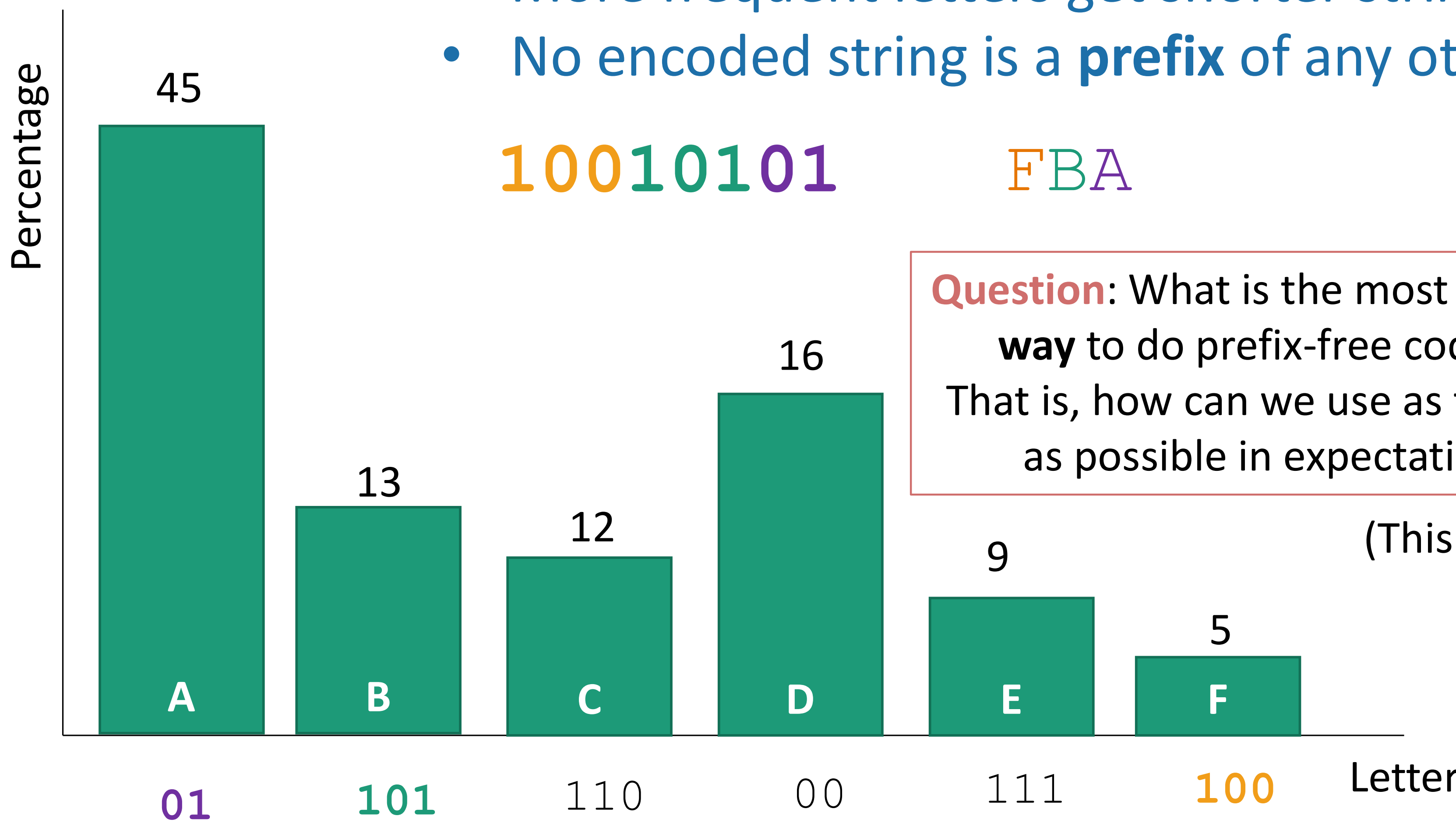
# Try 2: prefix-free coding

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

10010101 FBA

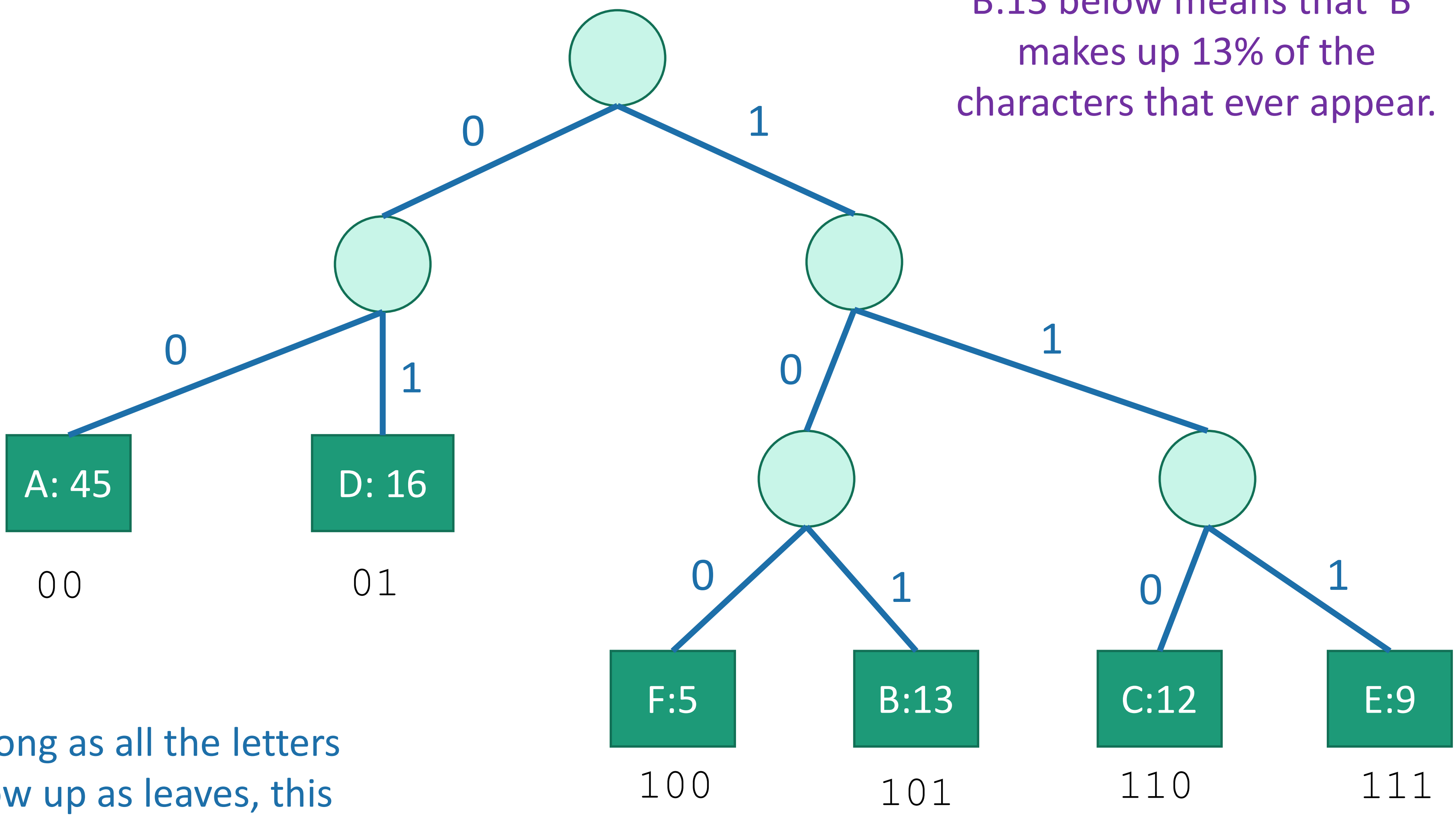
**Question**: What is the most **efficient way** to do prefix-free coding? That is, how can we use as few bits as possible in expectation?

(This is not it).



# A prefix-free 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 code is **prefix-free**.

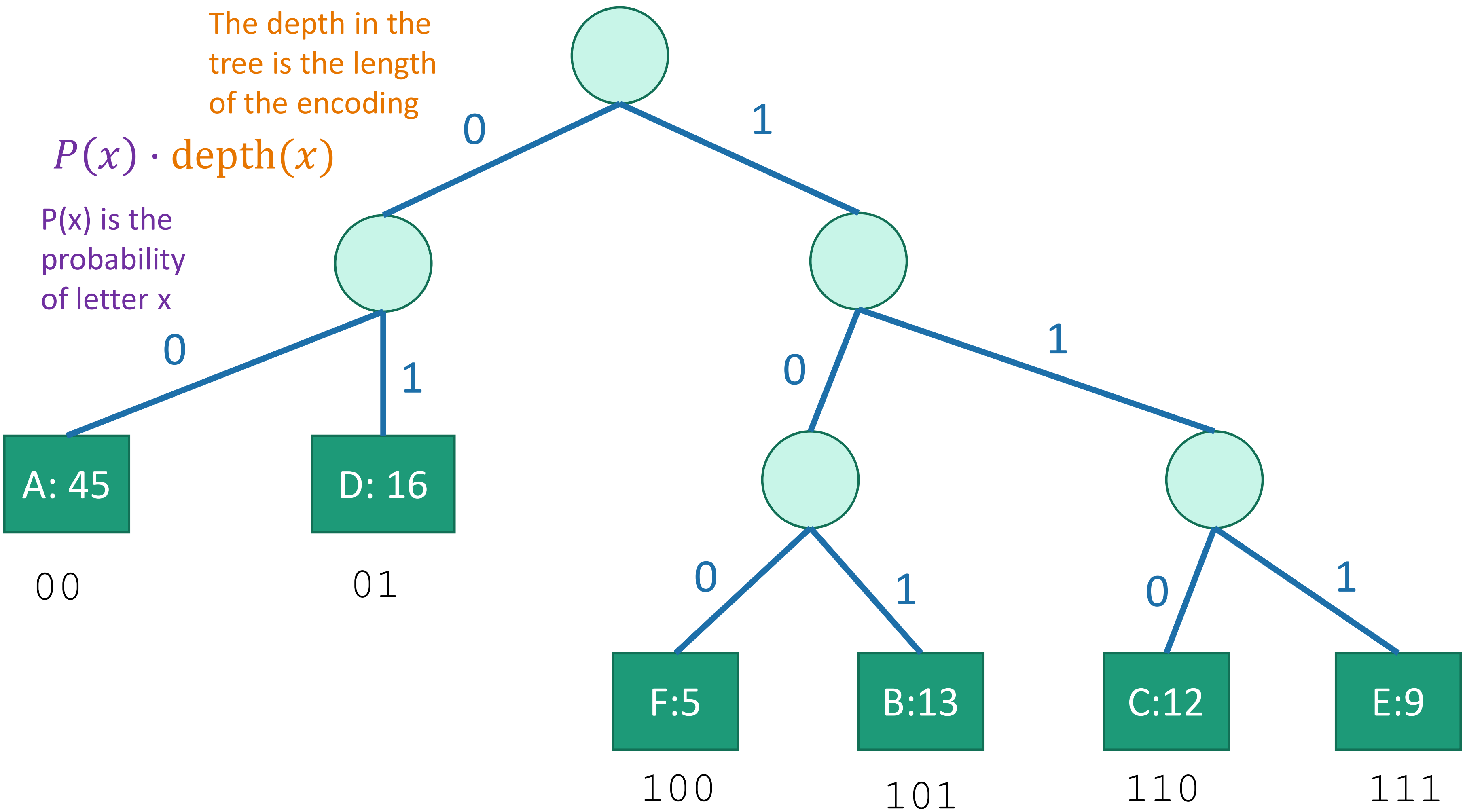
# How good is a tree?

- Imagine choosing a letter at random from the language.
  - Not uniformly random, but according to our histogram!
- The **cost of a tree** is the expected length of the encoding of a random letter.

Cost = $\sum_{leaves\ x} P(x) \cdot \mathrm{depth}(x)$

The depth in the tree is the length of the encoding

P(x) is the probability of letter x




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, where

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

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

# Greedy algorithm

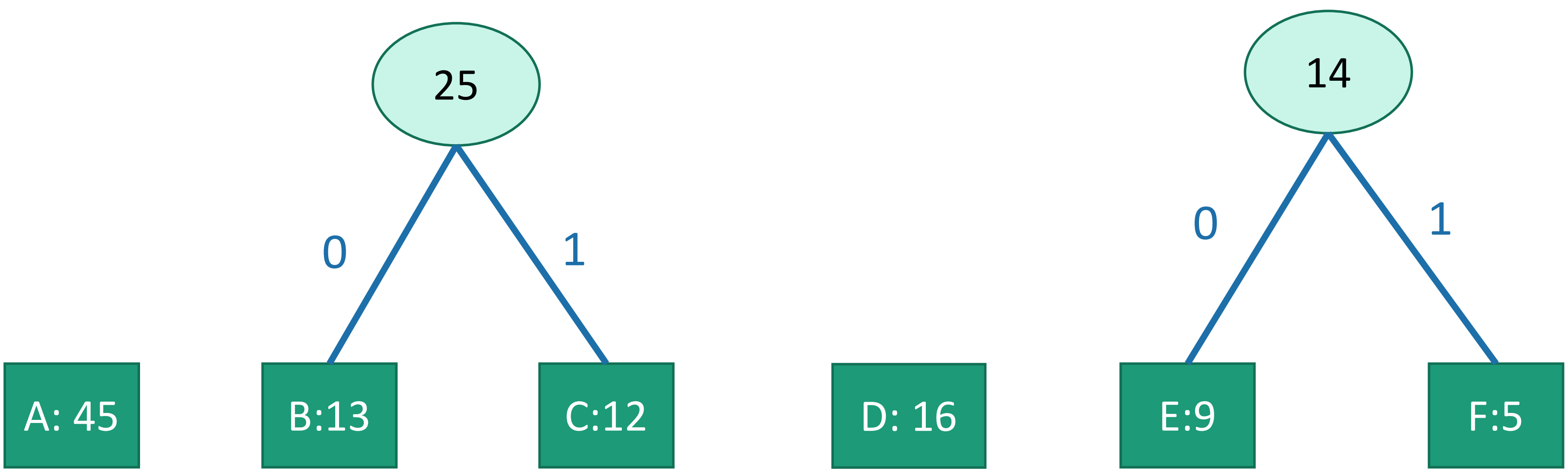
- Greedily build sub-trees from the bottom up.
- Greedy goal: less frequent letters should be further down the tree.

# Solution

greedily build subtrees, starting with the infrequent letters

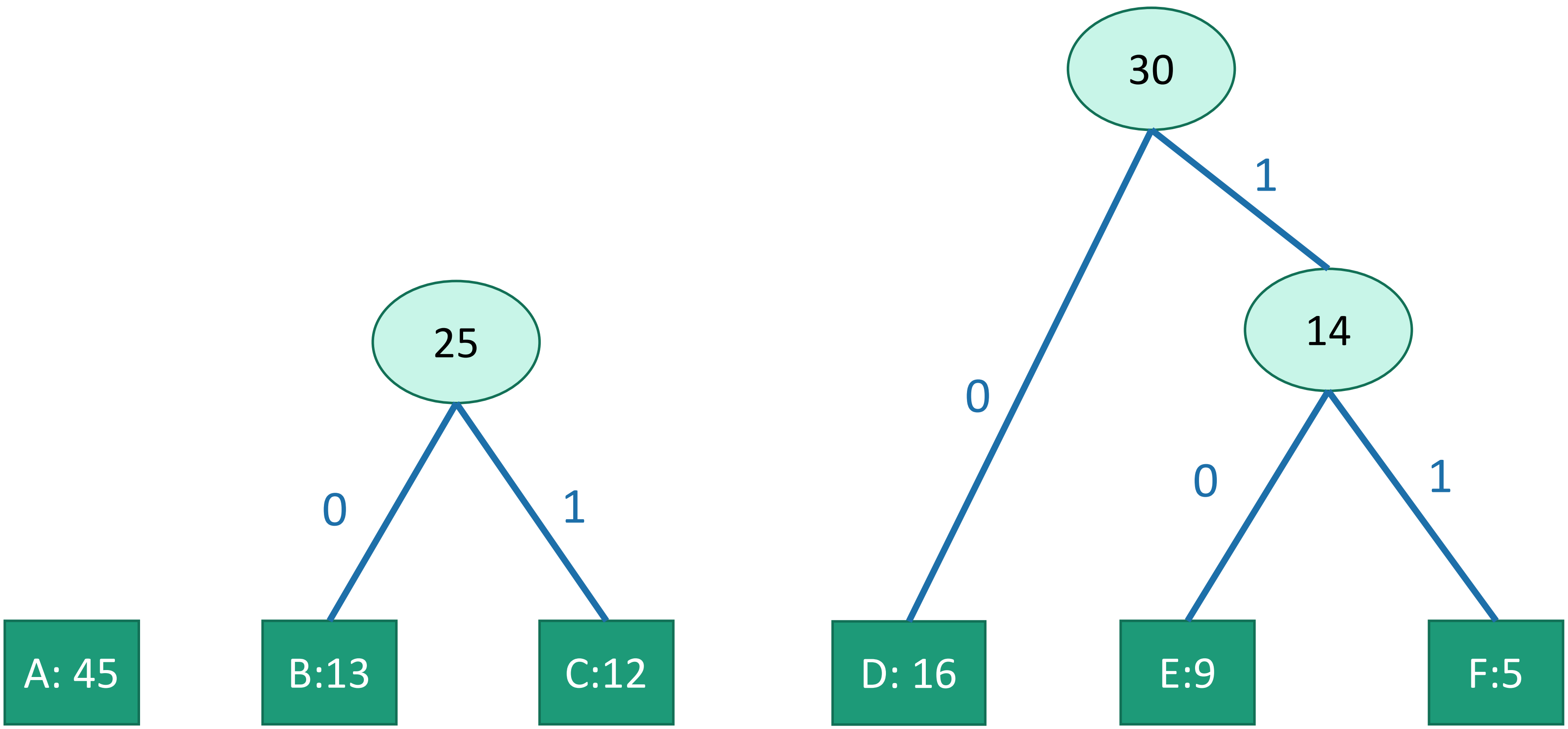
# Solution

## greedily build subtrees, starting with the infrequent letters



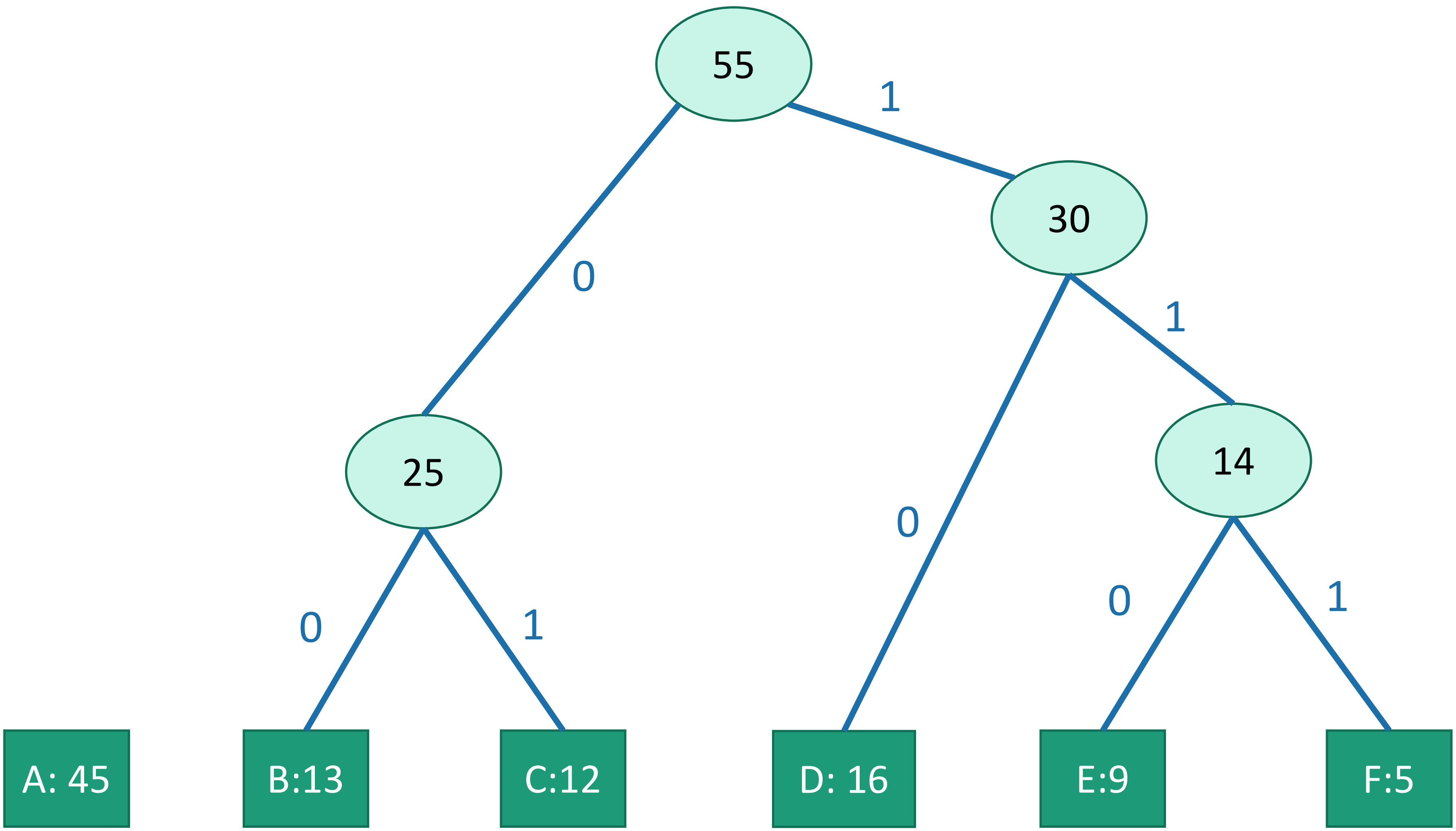
# Solution

## greedily build subtrees, starting with the infrequent letters



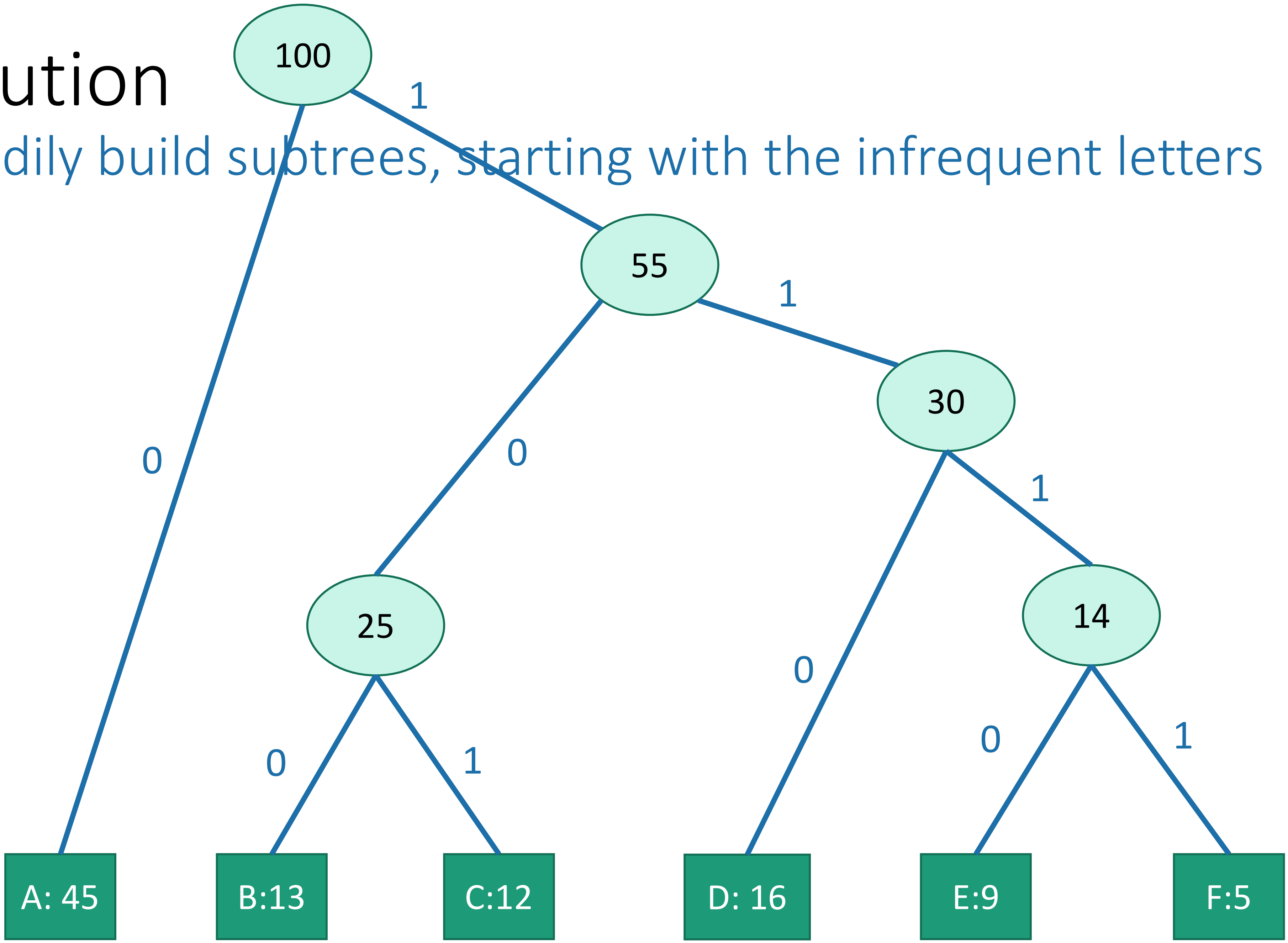
# Solution

greedily build subtrees, starting with the infrequent letters



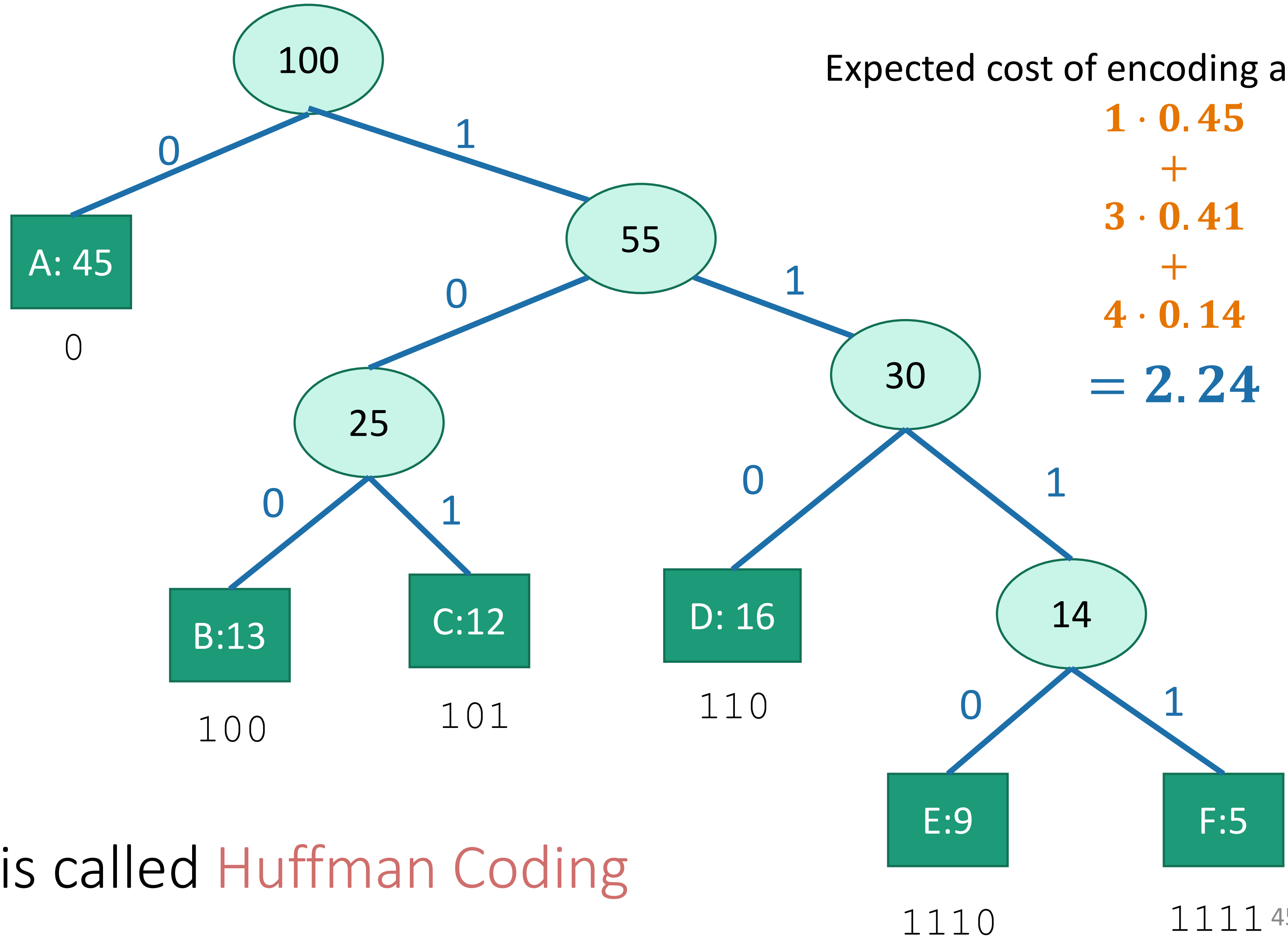
# Solution

greedily build subtrees, starting with the infrequent letters



# Solution

greedily build subtrees, starting with the infrequent letters




Expected cost of encoding a letter:

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

This is called Huffman Coding

# Tổng kết

- 3 ví dụ về giải thuật tham lam
- Dạng thuật toán dễ hiểu, dễ cài đặt
- Không phải lúc nào cũng đưa về được giải thuật tham lam
- Dạng bài toán phù hợp là:
  - Có cấu trúc con tối ưu dạng tuyến tính
    - Lời giải của bài toán con chỉ phụ thuộc vào một bài toán con nhỏ hơn

# Recap II

- Often easy to write down
- The natural greedy algorithm may not always be correct.
- A problem is a good candidate for a greedy algorithm if:
  - it has optimal substructure
  - that optimal substructure is **REALLY NICE**
    - solutions depend on just one other sub-problem.