Design and Analysis of Algorithms

NUS
National University of Singapore

CS3230

Week 7

Greedy Algorithms

Steven Halim

Chang Yi-Jun

Dynamic Programming (DP) algorithm paradigm

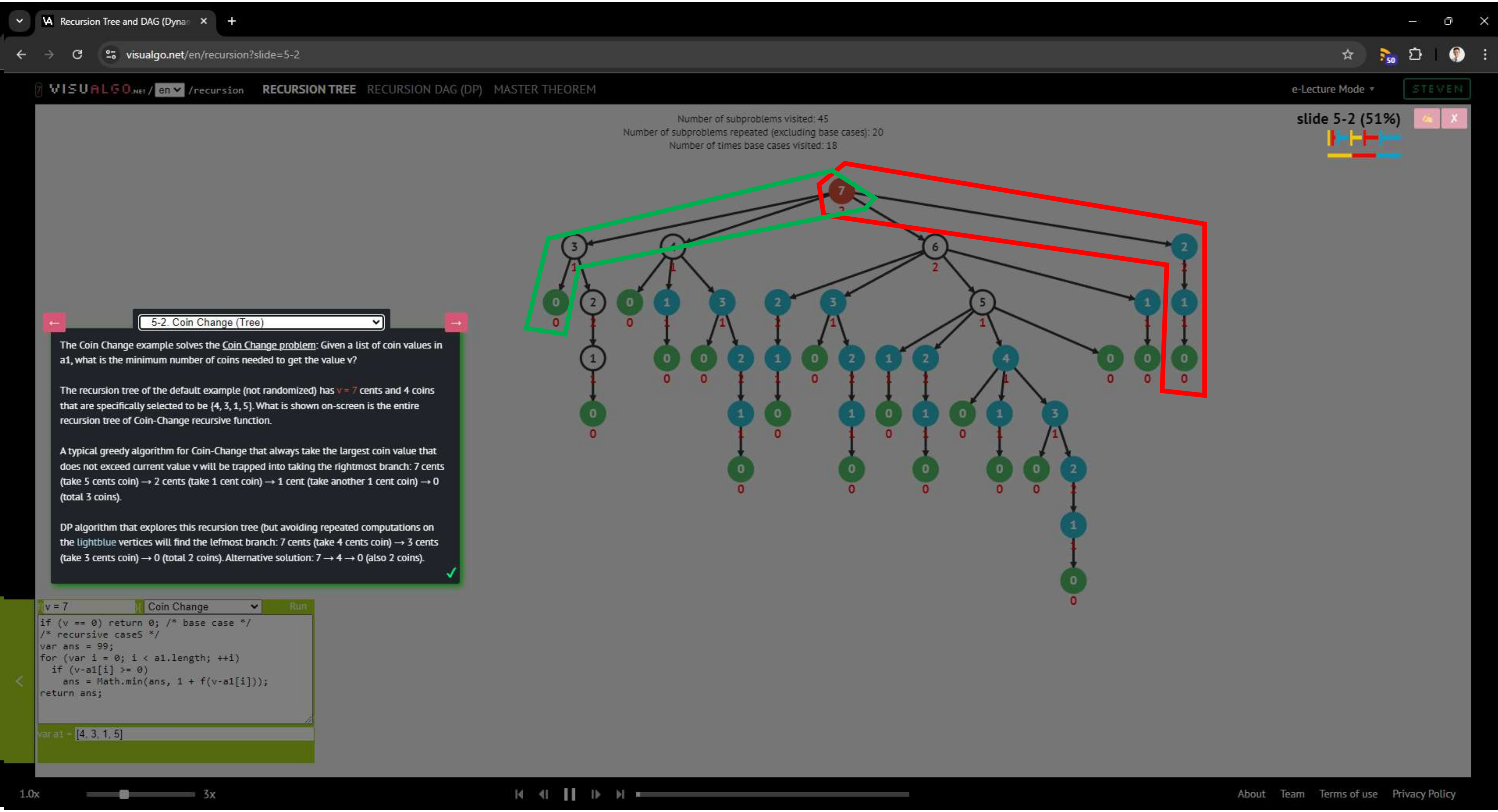
- Expressing the solution recursively
- Overall, there are only small (e.g., polynomial) number of subproblems
- But there is a huge overlap among the subproblems.
  So, the recursive algorithm may take exponential time
  (solving the same subproblem multiple times)
- So, we compute the recursive solution iteratively in a bottom-up fashion or recursively (top-down) but with memoization.
  This avoids wastage of computation → an efficient implementation

Today: Greedy Algorithms

A very general technique, like complete search (brute force), Divide-and-Conquer (D&C), and Dynamic Programming (DP)

The technique is to recast the problem so that
<u>only one subproblem</u> needs to be solved at each step.
It beats complete search, D&C, and DP,
when it works*.

*many times it does **not**…
Knowing when greedy is applicable for a given computational problem is the key skill


Recursion Tree and DAG (Dynan
visualgo.net/en/recursion?slide=5-2
/recursion
RECURSION TREE
RECURSION DAG (DP)
MASTER THEOREM
e-Lecture Mode
Number of subproblems visited: 45
Number of subproblems repeated (excluding base cases): 20
Number of times base cases visited: 18
slide 5-2 (51%)
5-2. Coin Change (Tree)
The Coin Change example solves the Coin Change problem: Given a list of coin values in a1, what is the minimum number of coins needed to get the value v?
The recursion tree of the default example (not randomized) has v = 7 cents and 4 coins that are specifically selected to be {4, 3, 1, 5}. What is shown on-screen is the entire recursion tree of Coin-Change recursive function.
A typical greedy algorithm for Coin-Change that always take the largest coin value that does not exceed current value v will be trapped into taking the rightmost branch: 7 cents (take 5 cents coin) → 2 cents (take 1 cent coin) → 1 cent (take another 1 cent coin) → 0 (total 3 coins).
DP algorithm that explores this recursion tree (but avoiding repeated computations on the lightblue vertices will find the lefmost branch: 7 cents (take 4 cents coin) → 3 cents (take 3 cents coin) → 0 (total 2 coins). Alternative solution: 7 → 4 → 0 (also 2 coins).
v = 7
Coin Change
Run
if (v == 0) return 0; /* base case */
/* recursive caseS */
var ans = 99;
for (var i = 0; i < a1.length; ++i)
if (v-a1[i] >= 0)
ans = Math.min(ans, 1 + f(v-a1[i]));
return ans;
var a1 = [4, 3, 1, 5]
1.0x
3x
About
Team
Terms of use
Privacy Policy

Recursion Tree and DAG (Dynar

visualgo.net/en/recursion

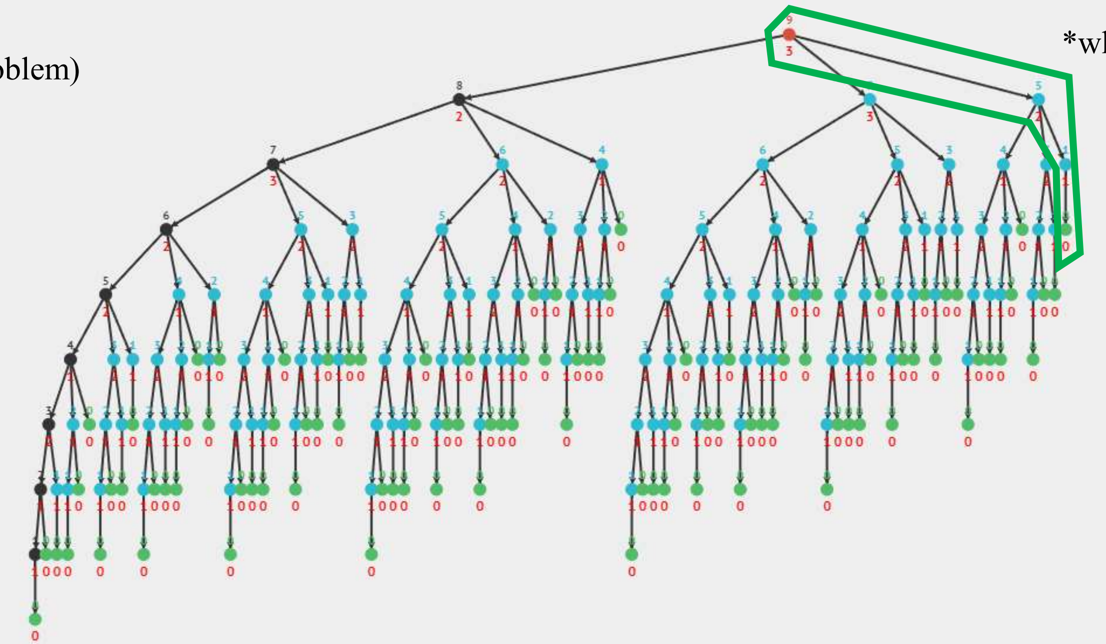
VISUALGO.NET / en /recursion RECURSION TREE RECURSION DAG (DP) MASTER THEOREM Exploration Mode STEVEN

Number of subproblems visited: 223
Number of subproblems repeated (excluding base cases): 118
Number of times base cases visited: 96

- Exponential if we try all
- O(v*k) if we use DP
  (avoiding recomputations of the same sub-problem)
- There are only 223-118-96 = 9
  distinct sub-problems for this screenshot

- O(v+k) if we use Greedy*

  *when applicable



v = 9 | Coin Change | Run

```
if (v == 0) return 0; /* base case */
/* recursive cases */
var ans = 99;
for (var i = 0; i < a1.length; ++i)
  if (v-a1[i] >= 0)
    ans = Math.min(ans, 1 + f(v-a1[i]));
return ans;
```

var a1 = [1, 2, 4]

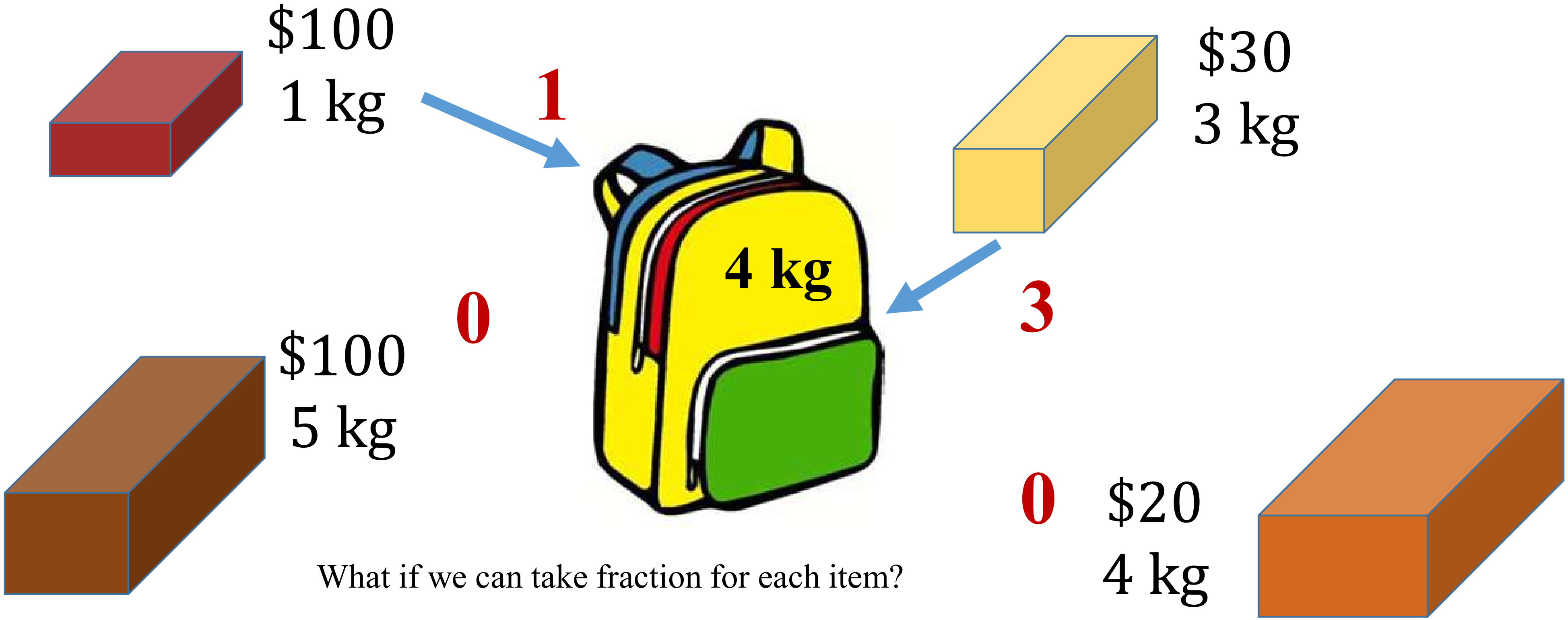
For some set of coin denominations, e.g., powers of twos {1, 2, 4}, being greedy (use the largest coin denomination that does not exceed v – this seems locally good, as it brings v down quickly), **is correct (it is really optimal) and much faster**

PS: On why this set of coins admits greedy strategy is not proven for this lecture

0.5x 3x About Team Terms of use Privacy Policy

0/1-Knapsack

For the 0/1-Knapsack version in DP lecture
(take the item or leave that item),
the optimal answer is 1 kg of \$100 + 3 kg of \$30 = \$130



What if we can take fraction for each item?

Fractional Knapsack

Input (identical to 0/1-Knapsack):

$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and W

Output:

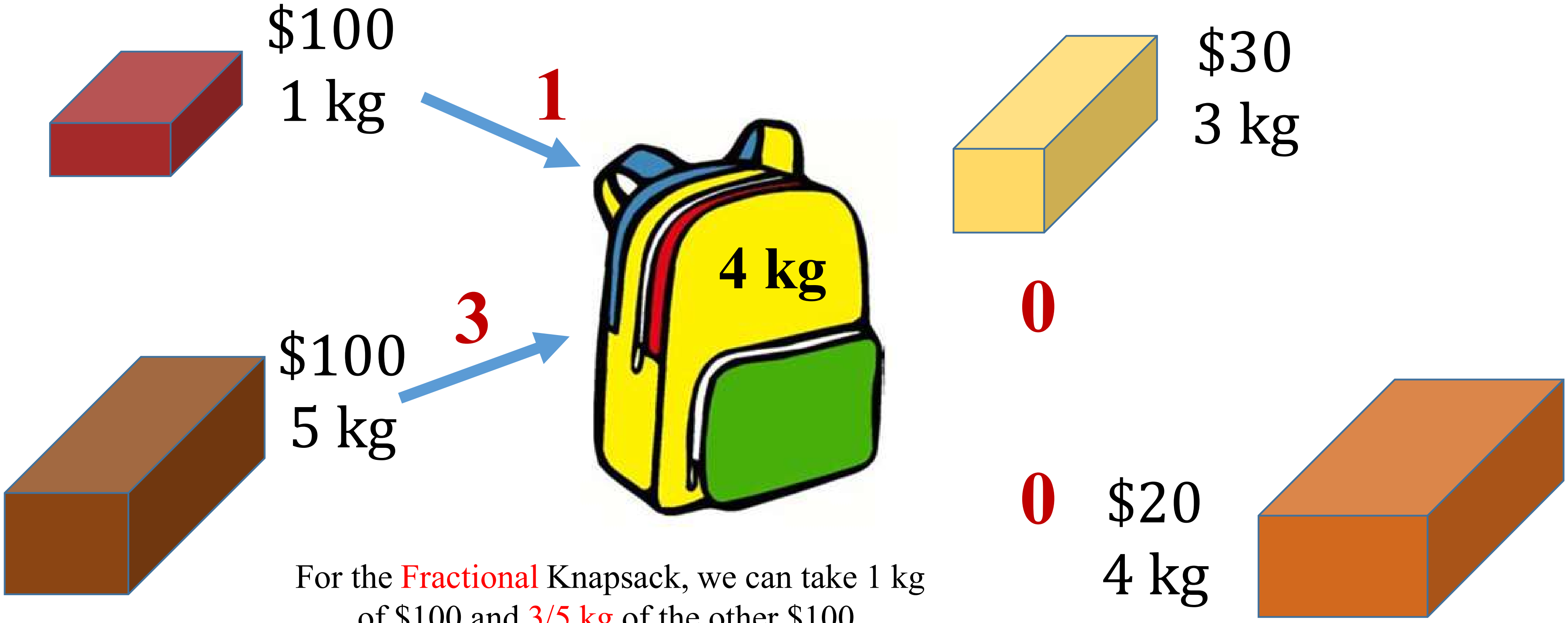
Compare this with 0/1-Knapsack version from DP lecture

Weights $x_1, x_2, \ldots, x_n$ that maximize $\sum_i v_i \cdot \frac{x_i}{w_i}$ subject to:

$$\sum_i x_i \leq W \text{ and } 0 \leq x_j \leq w_j \text{ for all } j \in [n].$$

Fractional Knapsack

For the 0/1-Knapsack version in DP lecture
(take the item or leave that item),
the optimal answer is 1 kg of \$100 + 3 kg of \$30 = \$130



For the Fractional Knapsack, we can take 1 kg
of \$100 and 3/5 kg of the other \$100,
for \$100+3/5*\$100 = \$160

Optimal Substructure

If we remove $\boldsymbol{y}$ **kgs of one item** $\boldsymbol{j}$ from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $\boldsymbol{W-y}$ **kgs** that one can take from the $\boldsymbol{n-1}$ **original items and** $\boldsymbol{w_j - y}$ **kgs of item** $\boldsymbol{j}$.

Optimal Substructure: Proof

cut-and-paste argument

- Let X be the value of an optimal knapsack.
- Suppose that the remaining load after removing y kgs of item j was not the optimal knapsack weighing at most $W - y$ kgs that one can take from the $n - 1$ original items and $w_j - y$ kgs of item j.
- This means that there is a(nother) knapsack of value $> X - v_j \cdot \frac{y}{w_j}$ with weight $\leq W - y$ kgs,
  among the $n - 1$ other items and $w_j - y$ kgs of item j.
- Combining with y kgs of item j gives knapsack of value $> X$ and weight at most W for original input.
- **Contradiction!**
  - **So the sub-structure must be optimal**

Dynamic Programming?

In the 0/1-Knapsack problem, we used the optimal substructure to formulate DP for deciding whether to add item j.

Then use O(nW) bottom-up (or top-down with memorization) solution.

But in this case, we can do better….

Question 1 at VA (Make a Guess)

Suppose you do not know anything about this problem before and you would like to solve the fractional knapsack problem in real life. What strategy will you use? Use your intuition.

1. Will first take the item with **maximum value**,
   then the item with **second maximum value**,
   and **so on** until the weight is exceeded (the last chosen item could be fractional)
2. Will first take the item with **minimum weight**,
   then the item with **second minimum weight**,
   and **so on** until the weight is exceeded (the last chosen item could be fractional)
3. Will first take the item with **maximum (value/weight)**,
   then the item with **second maximum (value/weight)**,
   and **so on** until the weight is exceeded (the last chosen item could be fractional)

Greedy-choice Property

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j.
Then, **there exists** an optimal knapsack containing $\min(w_{j^*}, W)$ kgs of item j^*.

Why? An "Exchange Argument":

- Suppose an optimal knapsack contains
  x_1 kgs of item 1, x_2 kgs of item 2, ..., x_n kgs of item n such that:
  $$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, W)$$
- Replace this weight by $\min(w_{j^*}, W)$ kgs of item j^*.
- Total weight does not change, and total value does not decrease because value/kg of j^* is maximum (sketch in the next slide).
- So, knapsack stays optimal, and it is "safe" to use this greedy-choice

Strategy for Greedy Algorithm

- Use greedy-choice property to put $\min(w_{j^*}, W)$ kgs of item j^* in knapsack.

- If knapsack now weighs W kgs, we are done.

- Otherwise, use optimal substructure to solve subproblem where all of item j^* is removed and knapsack weight limit is now $W - w_{j^*}$.

Iterative greedy algorithm

ITER-FRAC-KNAPSACK(v, w, W):

$valperkg \leftarrow [1,2,\ldots,n]$

Sort $valperkg$ using comparison operator $\preccurlyeq$ where $i \preccurlyeq j$ if $\frac{v[i]}{w[i]} \leq \frac{v[j]}{w[j]}$

for $i = n$ to 1: // O(n), back to front (largest ratio to smallest ratio)

if $W == 0$: **break**

$j \leftarrow valperkg[i]$

$k \leftarrow \min(w[j], W)$

print "k kgs of item j"

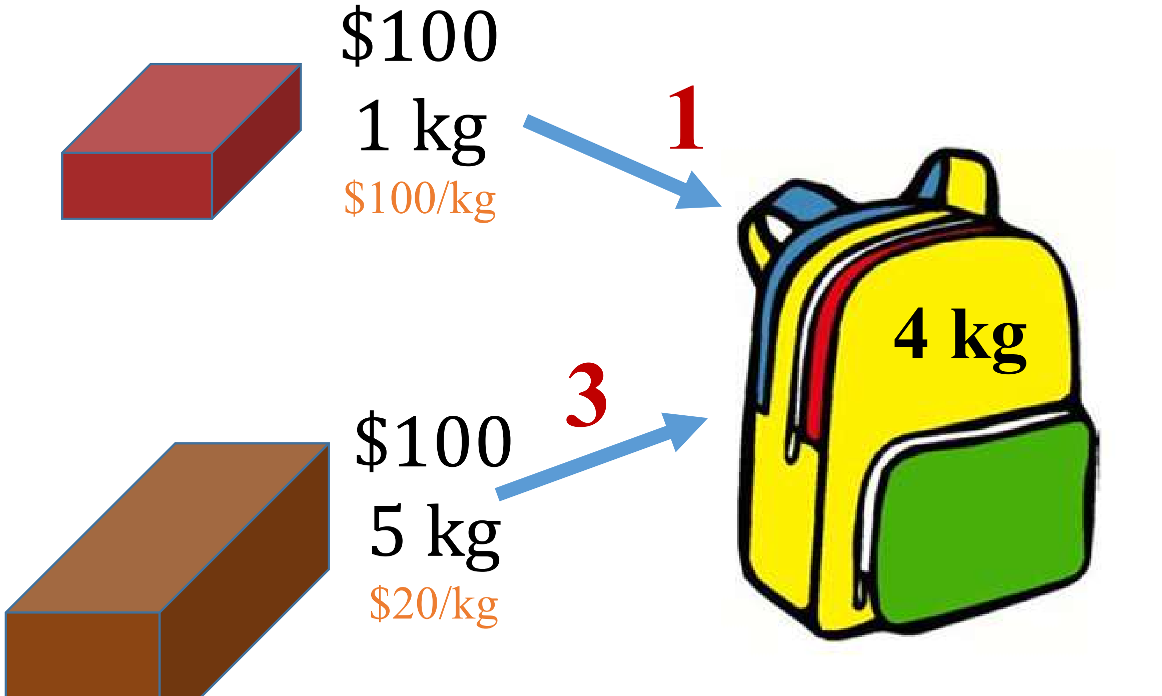
$W \leftarrow W - k$

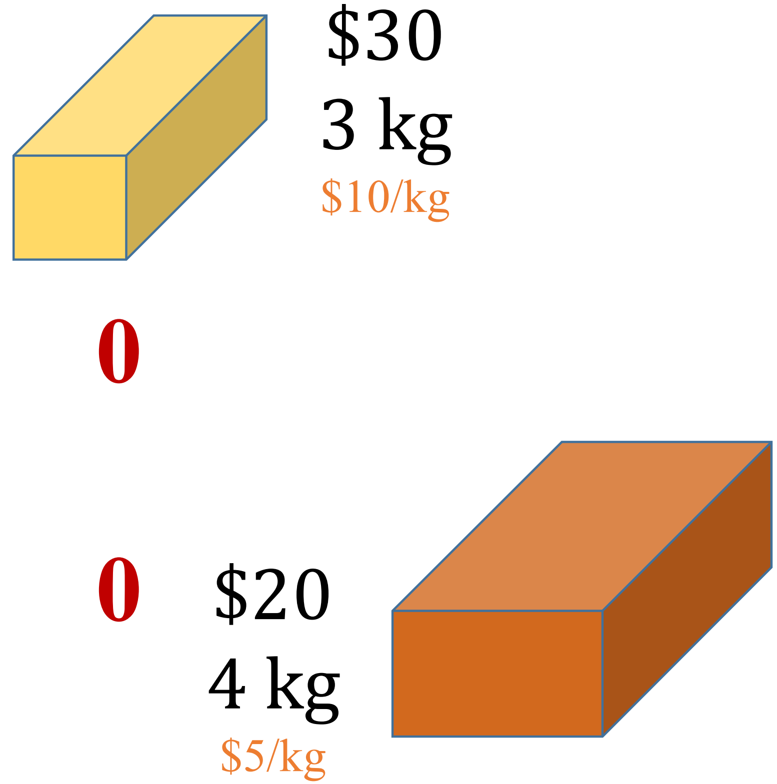
return

Total time in $\boldsymbol{O(n \log n)}$ due to sorting

Fractional Knapsack

For the 0/1-Knapsack version in DP lecture
(take the item or leave that item),
the optimal answer is 1 kg of $100 + 3 kg of $30 = $130





For the Fractional Knapsack, we can take 1 kg
of $100 and 3/5 kg of the other $100,
for $100+3/5*$100 = $160

Paradigm for greedy algorithms

1. Cast the problem where we must make a choice and are left with just one subproblem to solve.
2. Prove (exchange argument) that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe.
3. Use optimal substructure (cut and paste) to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem.

PS: You have seen more greedy algorithms before, i.e., Dijkstra's (for weighted SSSP), Prim's/Kruskal's (for MST)

Before Lecture Break

- There will be a few midterm-related announcement and/or tips
- All done verbally
- Review the recording if you do not attend the lecture

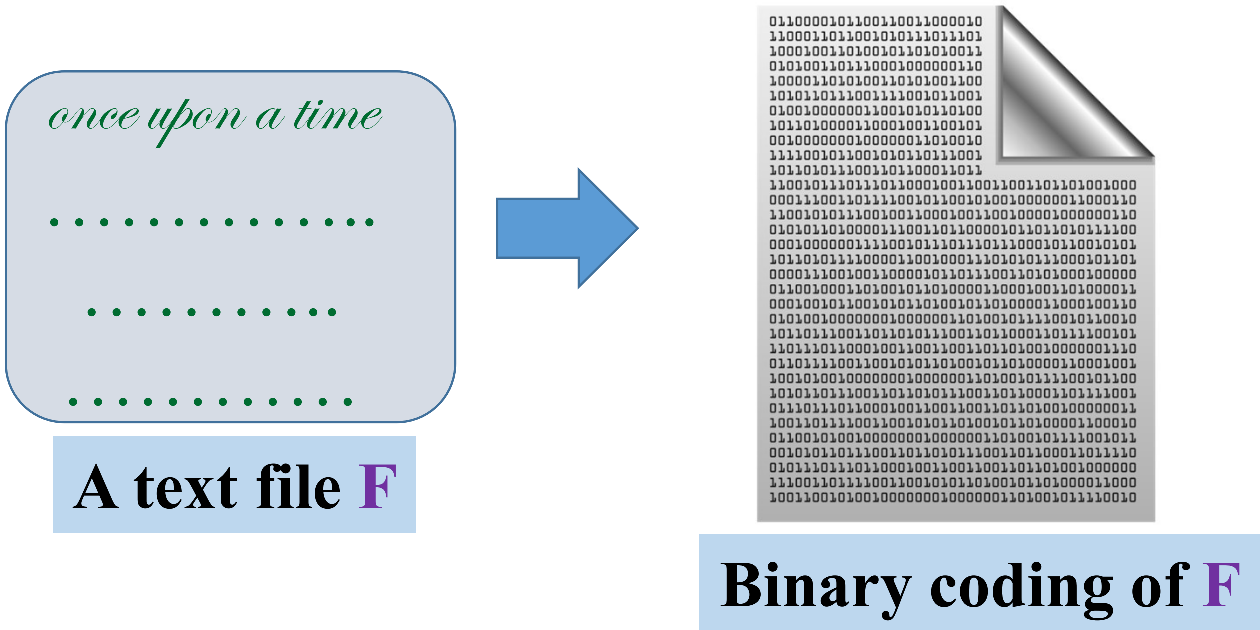
Huffman Code

Applications in data compression…

Binary coding

Alphabet set A : $\{a_1, a_2, \ldots, a_n\}$

A text File: a sequence of alphabets



Question: How many bits needed to encode a text file with m characters?

Answer: $m \lceil \log_2 n \rceil$ bits.

Fixed length encoding (1)

Alphabet set $\boldsymbol{A}$: $\{\boldsymbol{a_1}, \boldsymbol{a_2},\ldots, \boldsymbol{a_n}\}$

Question: What is a binary coding of $\boldsymbol{A}$?

Answer: $\boldsymbol{\gamma}$: $\boldsymbol{A}$ →**binary strings (PS: $\boldsymbol{\gamma}$ is read as 'upsilon')**

Question: What is a **fixed length** coding of $\boldsymbol{A}$?

Answer: each alphabet ← a unique binary string of length $\lceil \log_2 \boldsymbol{n} \rceil$.

Question: How to decode a **fixed length binary** coding?

Answer: **Easy ☺, suppose each has fixed-length of 4 bits**

$\mathbf{0100}|\mathbf{1010}|\mathbf{0000}|\mathbf{1011}|\ldots$

Fixed length encoding (2)

Alphabet set $\boldsymbol{A}$: $\{\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_n}\}$

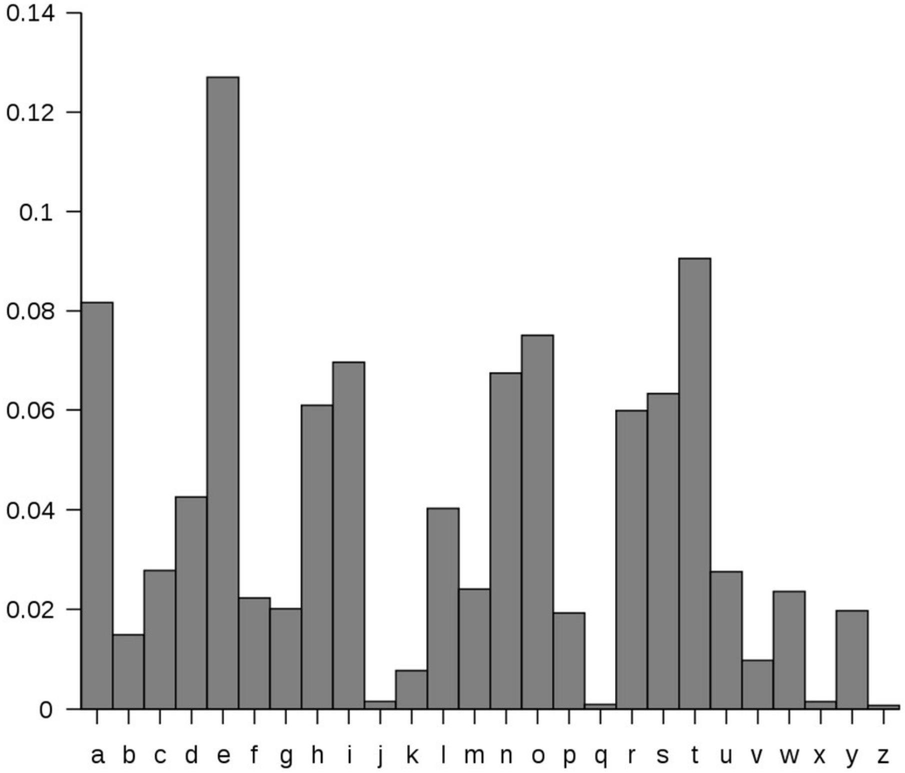
Question: Can we use fewer bits to store alphabet set $\boldsymbol{A}$?

Answer: No.

Question: Can we use fewer bits to store a file ?

Answer: **Yes**

Huge variation in the frequency of alphabets in a text (1)



ENGLISH LETTER FREQUENCIES
Per One-Thousand Letters

Sorted By Letter		Sorted By Frequency	
A	73	E	130
B	9	T	93
C	30	N	78
D	44	R	77
E	130	I	74
F	28	O	74
G	16	A	73
H	35	S	63
I	74	D	44
J	2	H	35
K	3	L	35
L	35	C	30
M	25	F	28
N	78	P	27
O	74	U	27
P	27	M	25
Q	3	Y	19
R	77	G	16
S	63	W	16
T	93	V	13
U	27	B	9
V	13	X	5
W	16	K	3
X	5	Q	3
		J	2
		Z	1

http://en.wikipedia.org/wiki/Letter_frequency

Huge variation in the frequency of alphabets in a text (2)

Question: How to exploit variation in the frequencies of alphabets ?

Answer (a.k.a., the 'greedy sense' / 'intuition'):

More frequent alphabets 🡐 coding with **shorter bit string**
Less frequent alphabets 🡐 coding with **longer bit string**

Variable length encoding (1)

Alphabets	Frequency f	Encoding γ
a	0.45	0
b	0.18	10
c	0.15	101
d	0.12	110
e	0.10	111

Average Bit Length per symbol using γ:

$$\mathbf{ABL}(\gamma) = \sum_{x \in A} f(x).\,|\gamma(x)|$$

$= \mathbf{0.45} \times \mathbf{1} + \mathbf{0.18} \times \mathbf{2} + (\mathbf{0.15} + \mathbf{0.12} + \mathbf{0.10}) \times \mathbf{3}$

$= \mathbf{1.92}$ (smaller than ceil($\log_2 5$) = 3 bits)

But there is a serious problem with the γ encoding. Can you see the issue?

Question: How will you decode 01010111 ?

Answer: $\boldsymbol{abbe}$ or $\boldsymbol{acae}$ ☹

Question: What is the source of this ambiguity?

Answer: $\gamma(\boldsymbol{b})$ is a prefix of $\gamma(\boldsymbol{c})$.

Question: Can you fix it?

Variable length encoding (2)

Alphabets	Frequency f	Encoding γ
a	0.45	0
b	0.18	100
c	0.15	101
d	0.12	110
e	0.10	111

Average **B**it **L**ength per symbol using γ:

$$\mathbf{ABL}(\gamma) = \sum_{x \in A} f(x).\,|\gamma(x)|$$

$= \mathbf{0.45} \times \mathbf{1} + \mathbf{0.18} \times \mathbf{3} + (\mathbf{0.15} + \mathbf{0.12} + \mathbf{0.10}) \times \mathbf{3}$

$= \mathbf{2.1}$ (a bit more than 1.92, but still less than 3 bits)

Prefix Coding

Definition:

A coding $\boldsymbol{\gamma(A)}$ is called **prefix coding** if there does not exist $x, y \in \boldsymbol{A}$ such that

$$\boldsymbol{\gamma(x)} \text{ is } \textbf{prefix} \text{ of } \boldsymbol{\gamma(y)}$$

Algorithmic Problem: Given a set $\boldsymbol{A}$ of $\boldsymbol{n}$ alphabets and their frequencies, compute coding $\boldsymbol{\gamma}$ such that

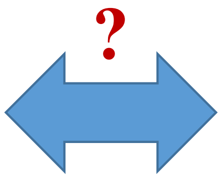
- $\boldsymbol{\gamma}$ is prefix coding
- $\mathbf{ABL}(\boldsymbol{\gamma})$ is **minimum**

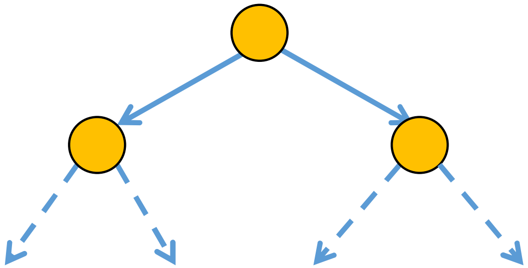
The challenge of the problem

Among all possible binary coding of ***A***,
how to find the **optimal prefix coding** ?

The novel idea of Huffman

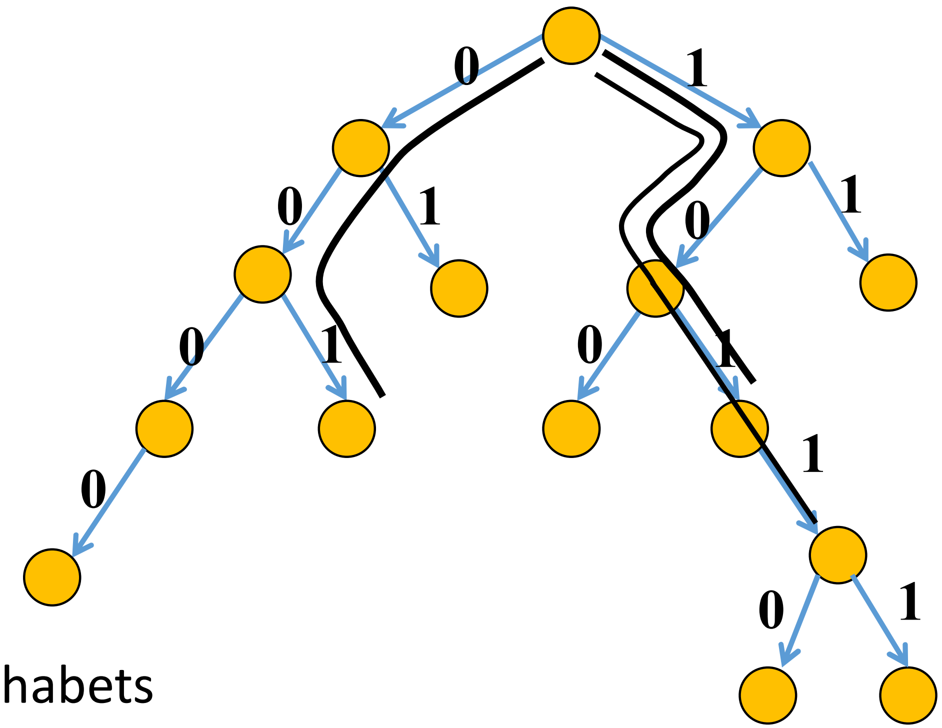
Binary coding





Binary tree

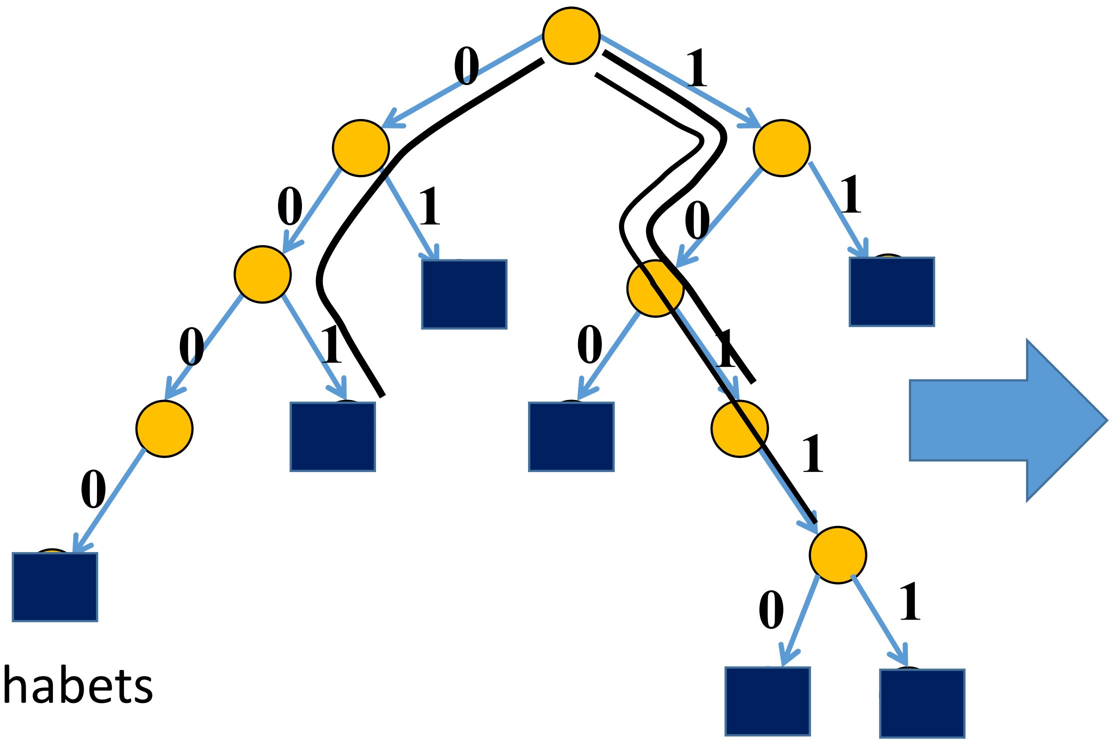
A labeled binary tree (1) – with animations



Leaf nodes → alphabets

Code of an alphabet = **Label of path from root**

A labeled binary tree (2) – with animations



01,
001,
0000,
11,
100,
10110,
10111

Leaf nodes → alphabets

Code of an alphabet = **Label of path from root**

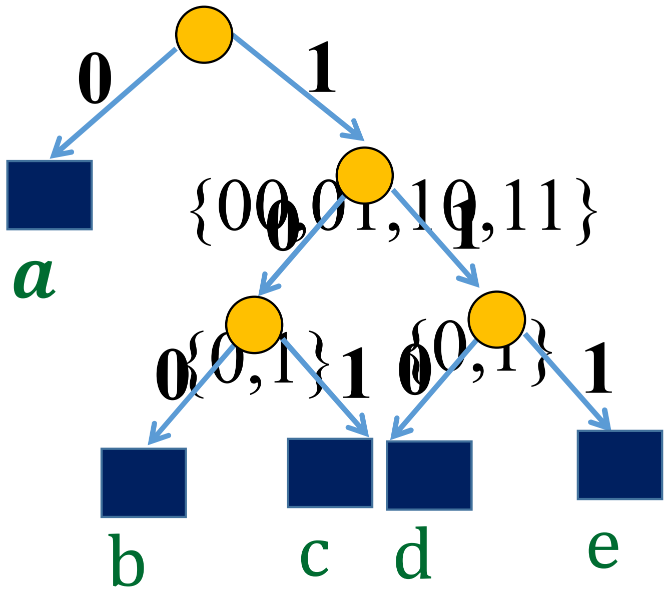
Variable length coding – with animations

Alphabets	Frequency f	Encoding γ
a	**0.45**	0
b	**0.18**	100
c	**0.15**	101
d	**0.12**	110
e	**0.10**	111

Question:

How to build the labeled tree for a prefix code ?

$\{0, 100, 101, 110, 111\}$



Prefix code and Labelled Binary tree

Theorem:

For each prefix code of a set $\boldsymbol{A}$ of $\boldsymbol{n}$ alphabets, there exists a binary tree $\boldsymbol{T}$ on $\boldsymbol{n}$ leaves s.t.

- There is a bijective (one to one) mapping between the **alphabets** and the **leaves.**
- The label of a path from root to a leaf node corresponds to the prefix code of the corresponding alphabet.

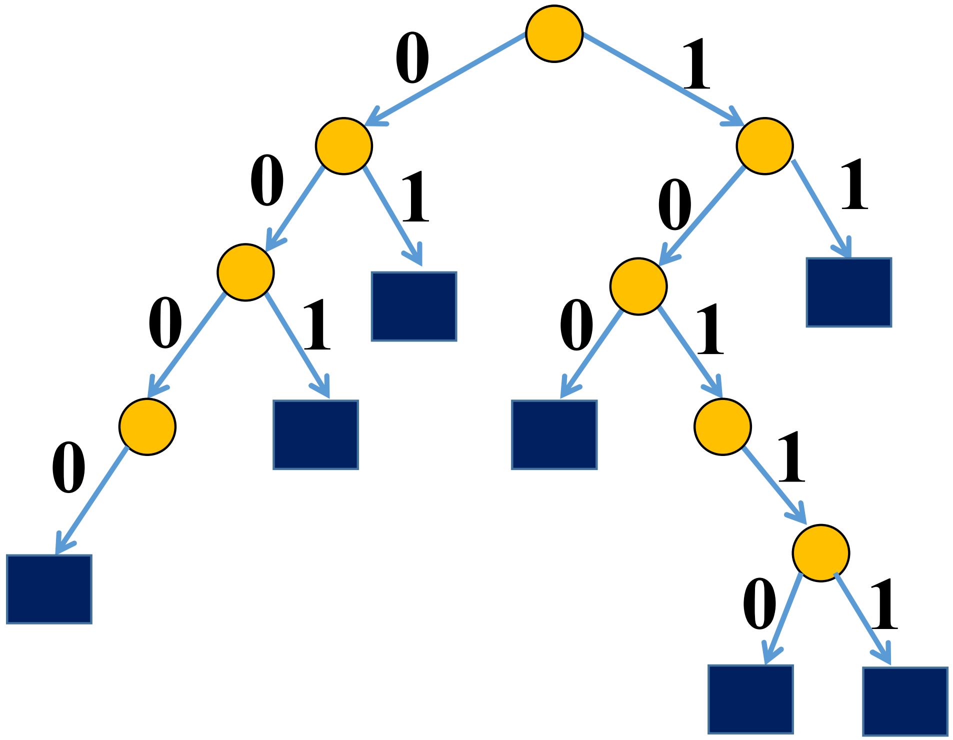
Question: Can you express **Average bit length** of $\boldsymbol{\gamma}$ in terms of its binary tree $\boldsymbol{T}$?

$$\mathbf{ABL}(\boldsymbol{\gamma}) = \sum_{x \in A} f(x).\,|\boldsymbol{\gamma}(x)|$$
$$= \sum_{x \in A} f(x).\,|\mathbf{depth_T}(x)|$$

PS: depthT is non-negative, the absolute symbol is not needed

Finding the labeled binary tree for an <u>optimal</u> prefix codes

Is the following prefix coding optimal ? – with animations



NO

Observations on the binary tree of an optimal prefix code

Lemma (not proven in this lecture):

The binary tree corresponding to optimal prefix coding must be a **full binary tree**:

Every internal node has degree exactly 2.

Question: What next ?

We need to see the influence of frequencies on an optimal binary tree.

Let $\boldsymbol{a_1}$, $\boldsymbol{a_2}$,…, $\boldsymbol{a_n}$ be the alphabets of $\boldsymbol{A}$ in non-decreasing order of their frequencies. So $\boldsymbol{a_1}$ is the *least frequent* alphabet.

Observations on the binary tree of an optimal prefix code

Intuitively, **more frequent** alphabets should be **closer to the root** and

less frequent alphabets should be **farther from the root**.

But how to organize them to achieve optimal prefix code ?

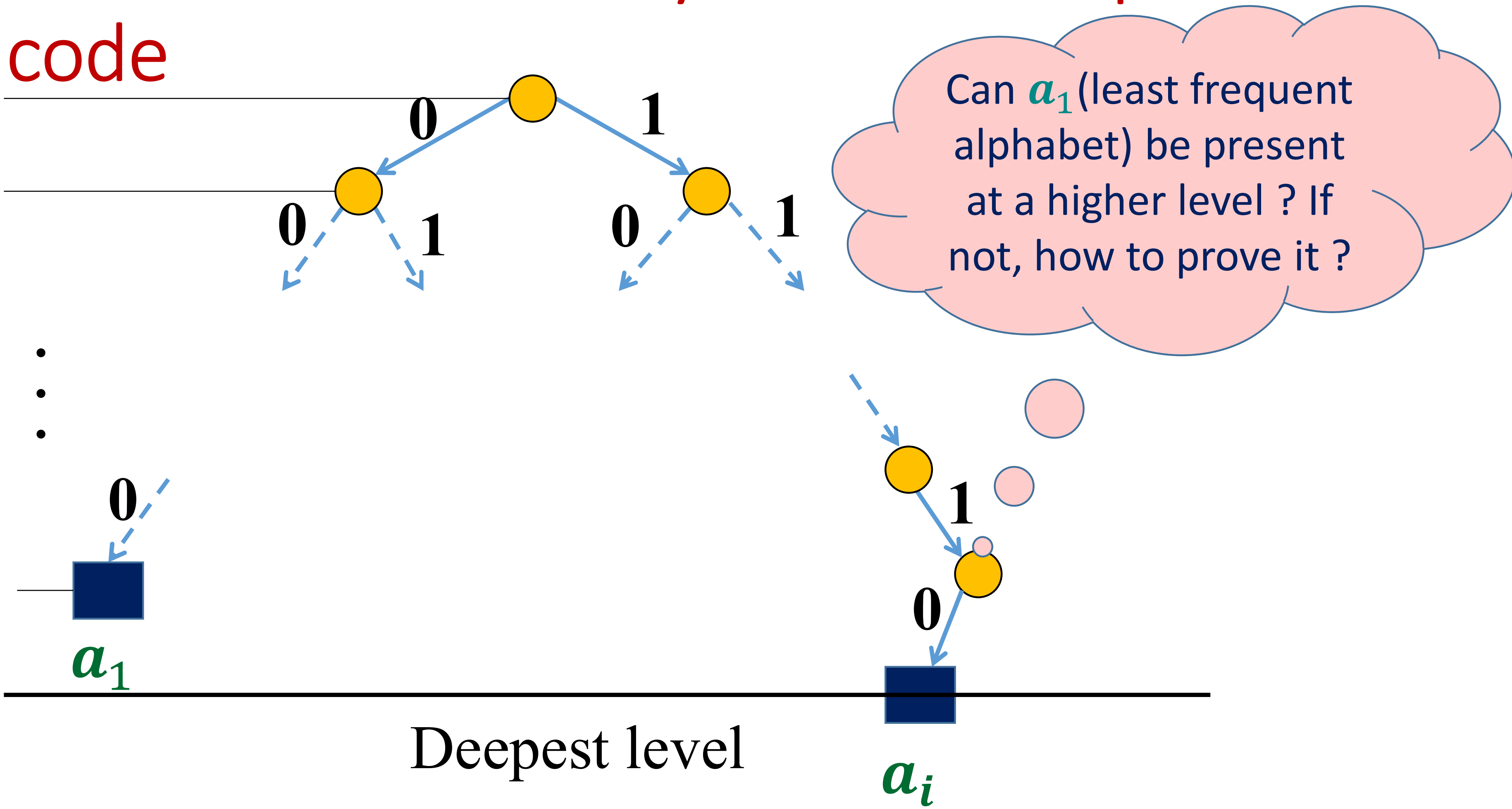
- We shall now make some simple observations about the structure of the binary tree corresponding to the optimal prefix codes.

- These observations will be about some local structure in the tree.

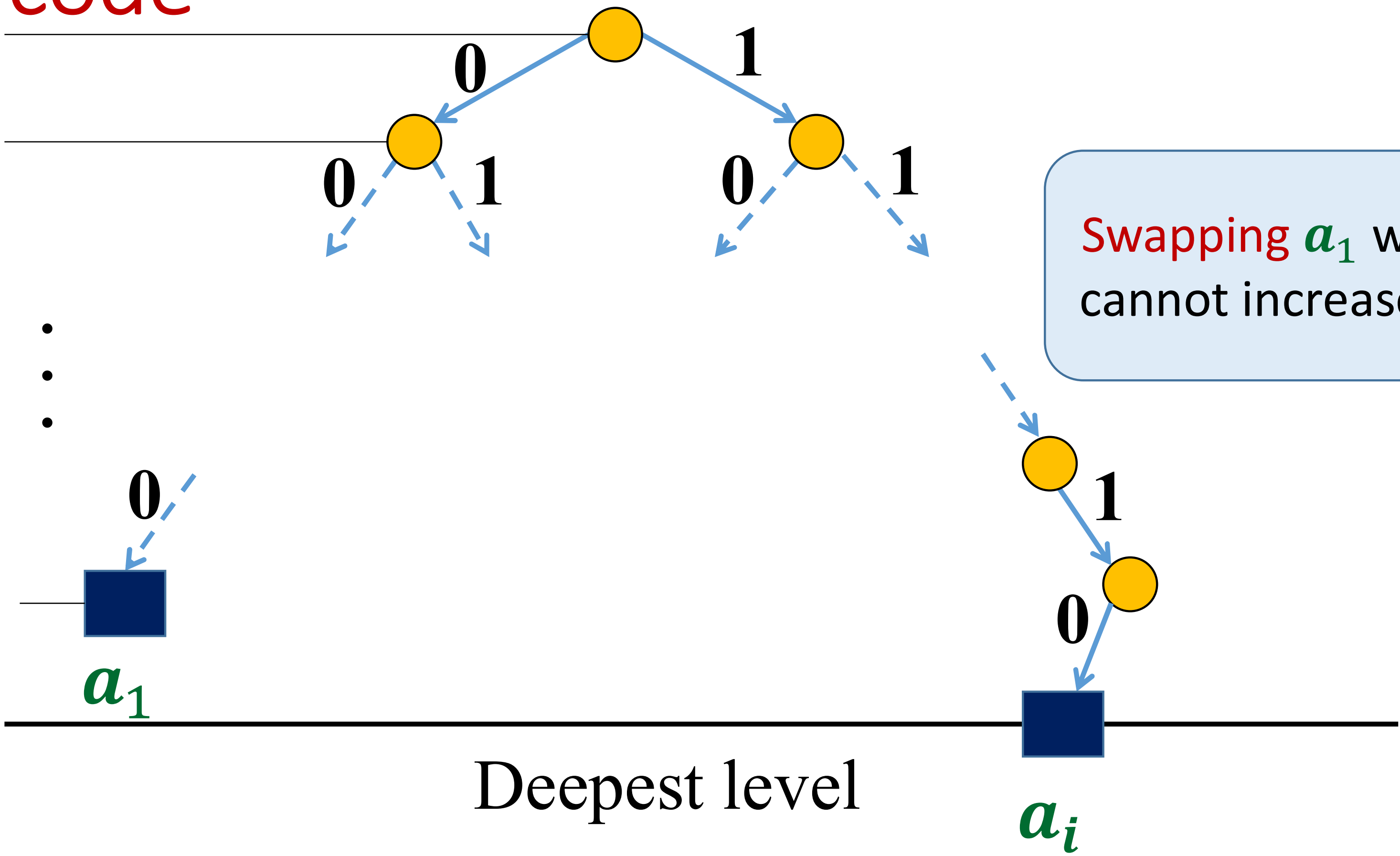
- Nevertheless, these observations will play a crucial role in the design of a binary tree with optimal prefix code for given $\boldsymbol{A}$.

Please pay full attention on the next few slides.

Observations on the binary tree of an optimal prefix code

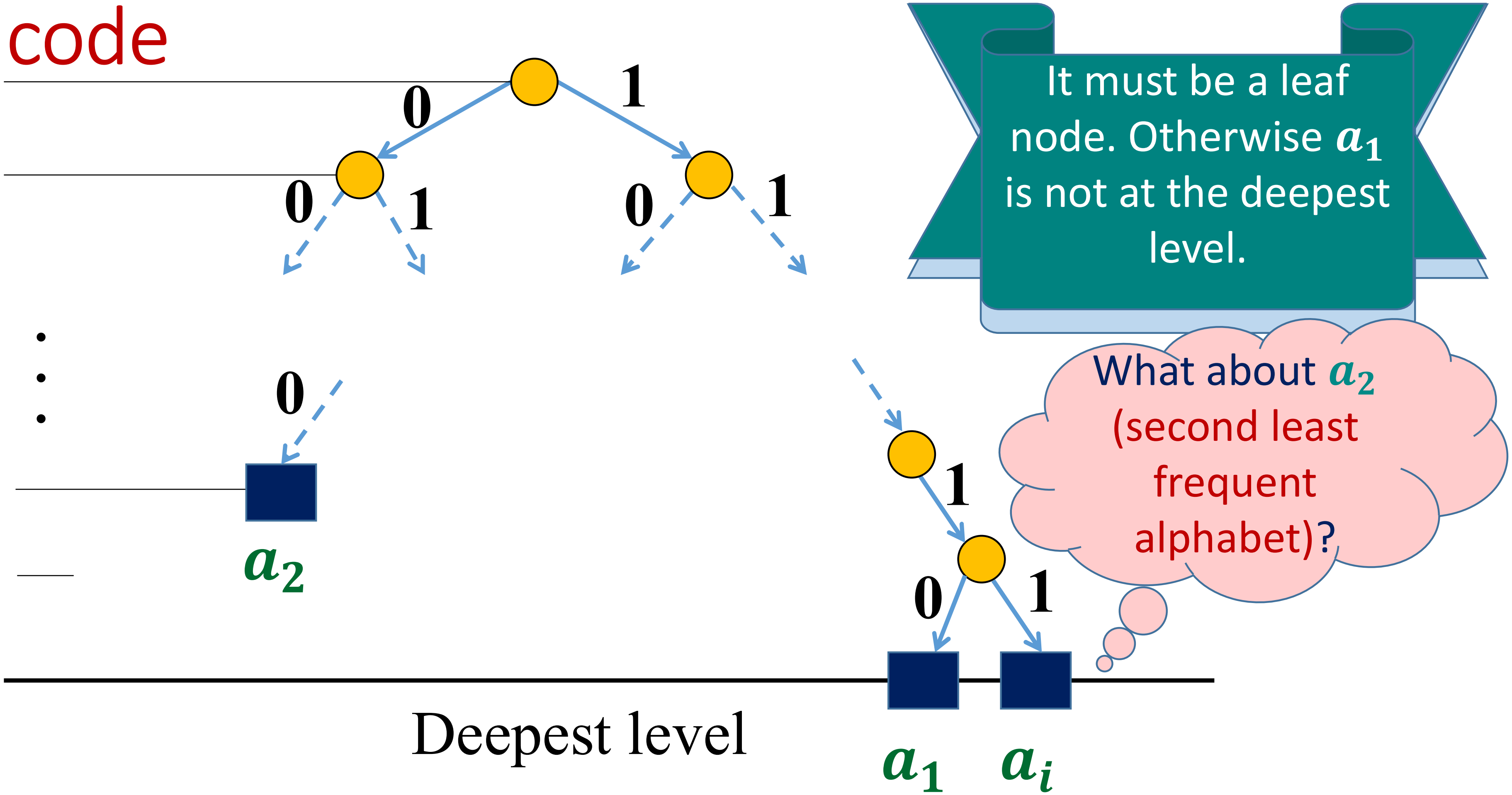


Observations on the binary tree of an optimal prefix code

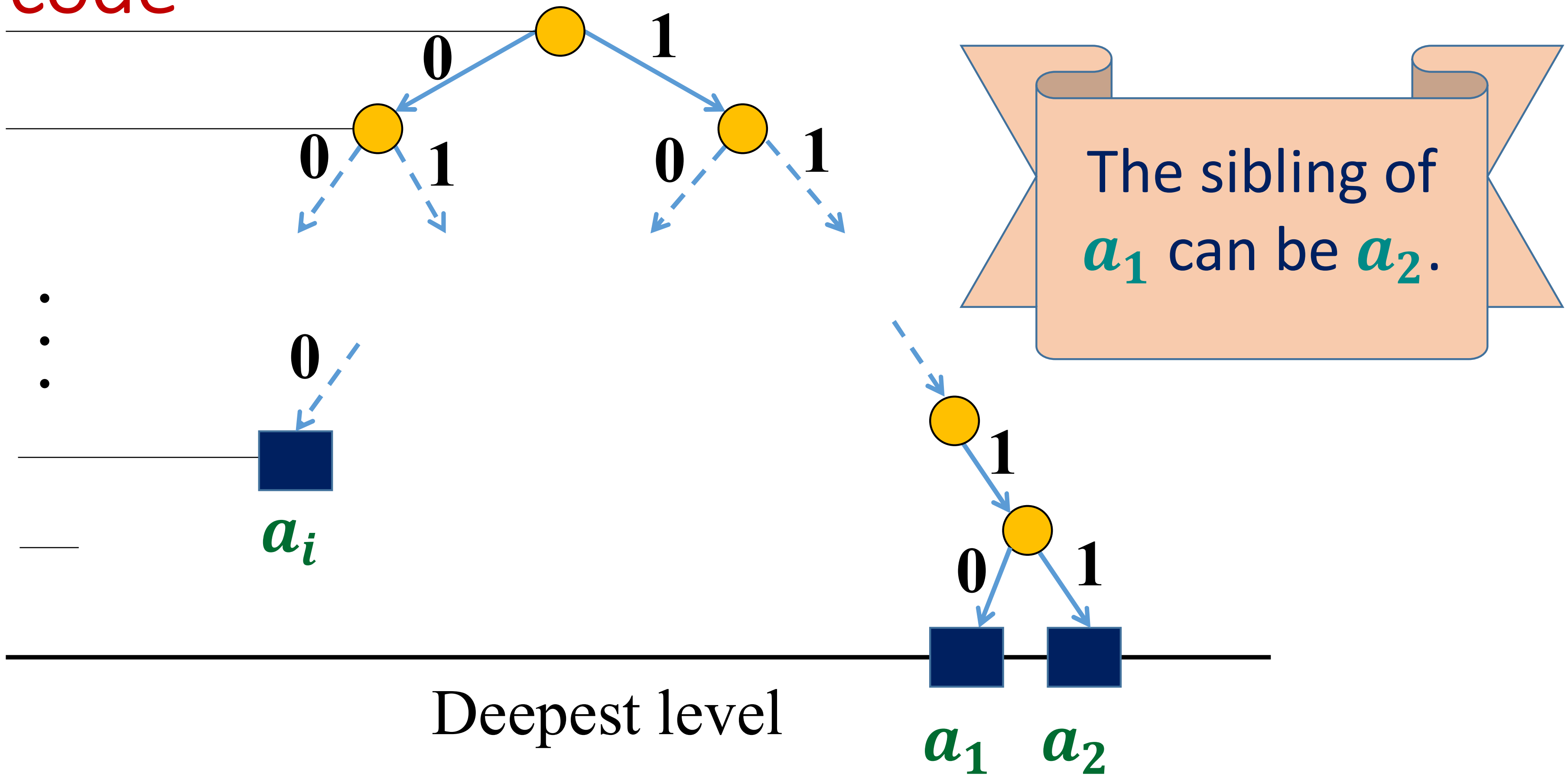


Swapping a_1 with a_i cannot increase **ABL**.

Observations on the binary tree of an optimal prefix code



Observations on the binary tree of an optimal prefix code



An important observation

Lemma: *There exists an optimal* prefix coding in which $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ appear as siblings in the corresponding labeled binary tree.

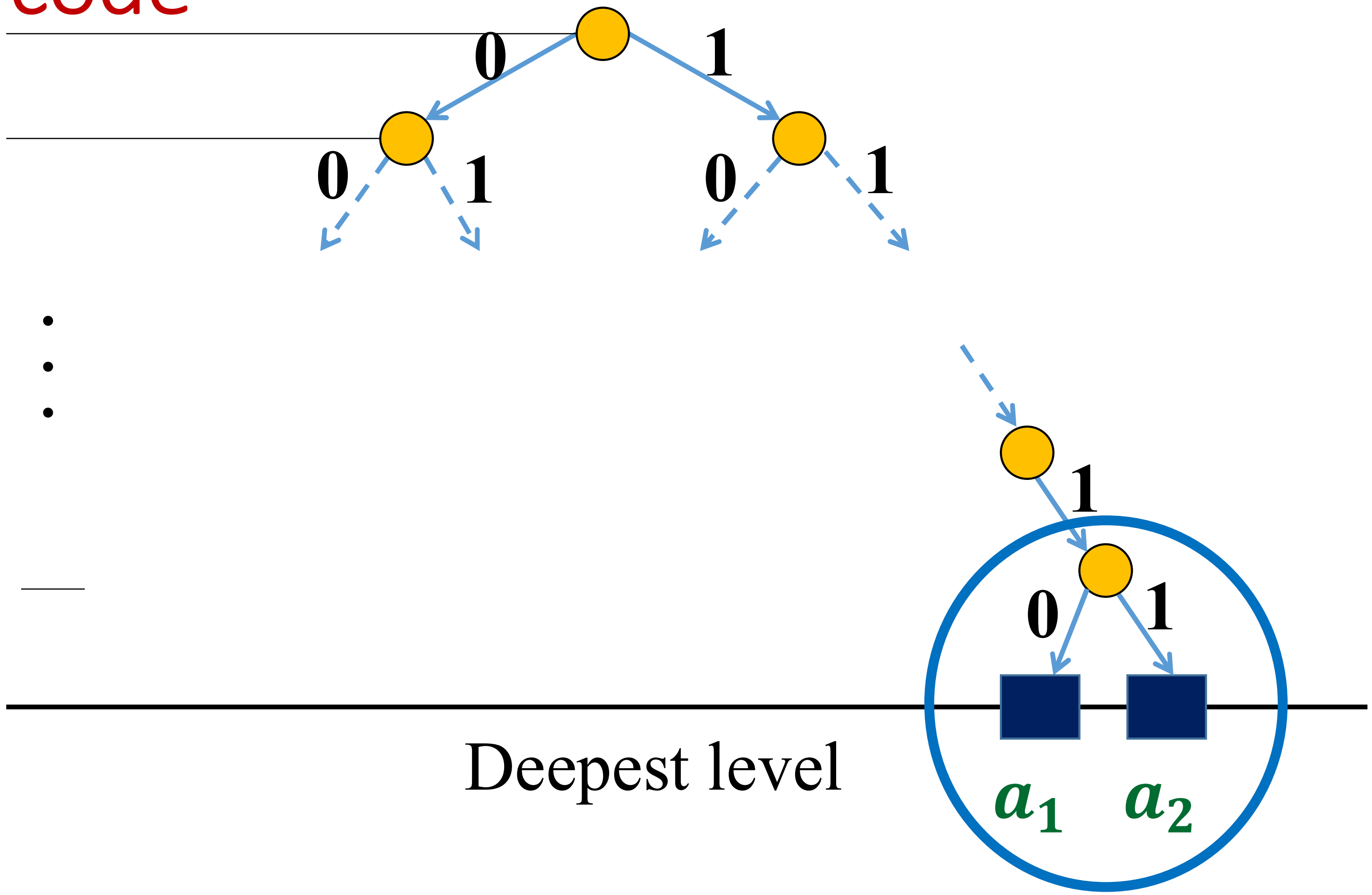
Important note: It is inaccurate to claim that "In every optimal prefix coding, $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ appear as siblings in the labeled binary string."

But algorithmic implication of the Lemma mentioned above is quite important:

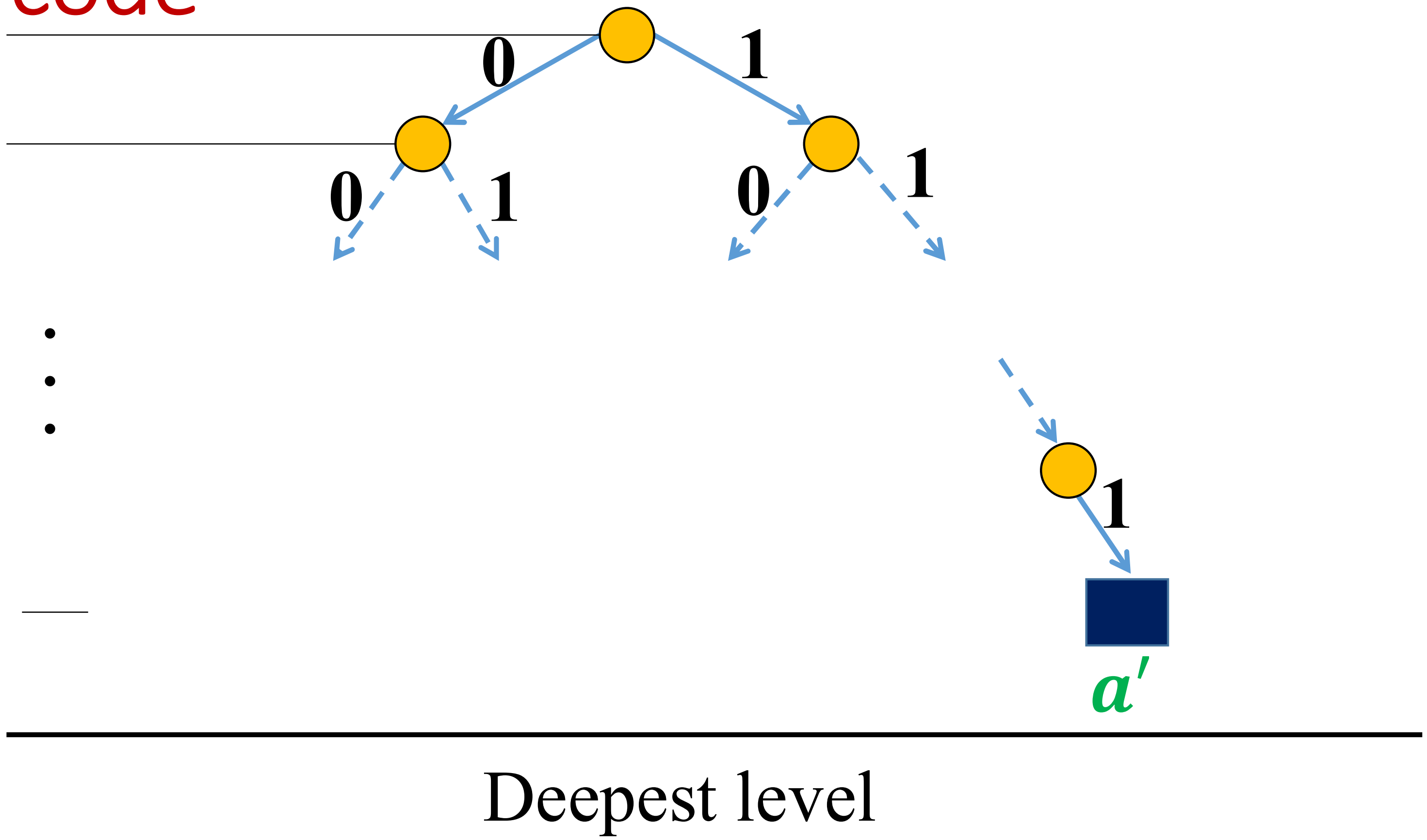
➔ We just need to focus on that binary tree of optimal prefix coding in which $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ appear as siblings.

This lemma is a powerful hint to the design of optimal prefix code.

Observations on the binary tree of an optimal prefix code



Observations on the binary tree of an optimal prefix code



Key Idea to design an algorithm (1)

$\boldsymbol{A} = \boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_n}$ be $\boldsymbol{n}$ alphabets in non-decreasing order of frequencies

$\Downarrow$

$\boldsymbol{A'} = \boldsymbol{a_3}, \ldots, \boldsymbol{a'}, \ldots, \boldsymbol{a_n}$ be $\boldsymbol{n-1}$ alphabets in non-decreasing order of frequencies with $\boldsymbol{f}(\boldsymbol{a'}) = \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$

Intuition (from the previous slide):

May be : An optimal prefix code of $\boldsymbol{A'}$ ➔ optimal prefix code of $\boldsymbol{A}$

Key Idea to design an algorithm (2)

Two notations:

- $\mathbf{OPT_{ABL}}(\boldsymbol{A})$: Minimum **ABL** value over all prefix code/labelled binary tree for alphabet A
- $\mathbf{OPT}(\boldsymbol{A})$: A prefix code/labelled binary tree for alphabet A with **ABL** value $\mathbf{OPT_{ABL}}(\boldsymbol{A})$

Recall, $\mathbf{ABL}(\boldsymbol{\gamma}) = \sum_{x \in \boldsymbol{A}} f(x).\,|\boldsymbol{\gamma}(x)| = \sum_{x \in \boldsymbol{A}} f(x).\,|\mathbf{depth_T}(x)|$

Key Idea to design an algorithm (3)

$\boldsymbol{A} = \boldsymbol{a_1}, \boldsymbol{a_2},\ldots, \boldsymbol{a_n}$ be $\boldsymbol{n}$ alphabets in non-decreasing order of frequencies

$\boldsymbol{A'} = \boldsymbol{a_3},\ldots, \boldsymbol{a'},\ldots, \boldsymbol{a_n}$ be $\boldsymbol{n-1}$ alphabets in non-decreasing order of frequencies with $\boldsymbol{f}(\boldsymbol{a'}) = \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$

Question: What should be the relation between $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A})$ and $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'})$?

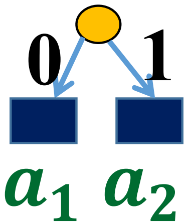
Answer: $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) + \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$

Observation: If this relation is true, we have an algorithm for optimal prefix codes.

The algorithm based on

$OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$

OPT(A)

{ **If** $|A|$=2, return 

;

else

{ Let a_1 and a_2 be the two alphabets with **least frequencies**.

Remove a_1 and a_2 from A;

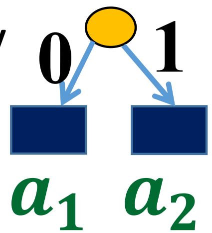
Create a new alphabet a';

$f(a') \leftarrow f(a_1) + f(a_2)$;

Insert a' into A;

$T \leftarrow$ OPT(A);

Replace node 

in T by 

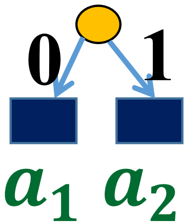
;

return T;

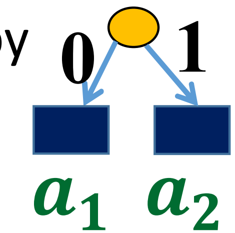
} }

The algorithm based on

$$OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$$

OPT(A)
{ If $|A|$=2, return 

;
else
{ Let a_1 and a_2 be the two alphabets with **least frequencies**.
Remove a_1 and a_2 from A;
Create a new alphabet a';
$f(a') \leftarrow f(a_1) + f(a_2)$;
Insert a' into A;
$T \leftarrow$ OPT(A);
Replace node 

in T by 

;
return T ;
} }

Build a heap for the alphabets with frequencies as key takes $O(n \log n)$ time

Perform **Remove** and **Insert** in $O(\log n)$ time

Overall time = $O(n \log n)$

How to prove

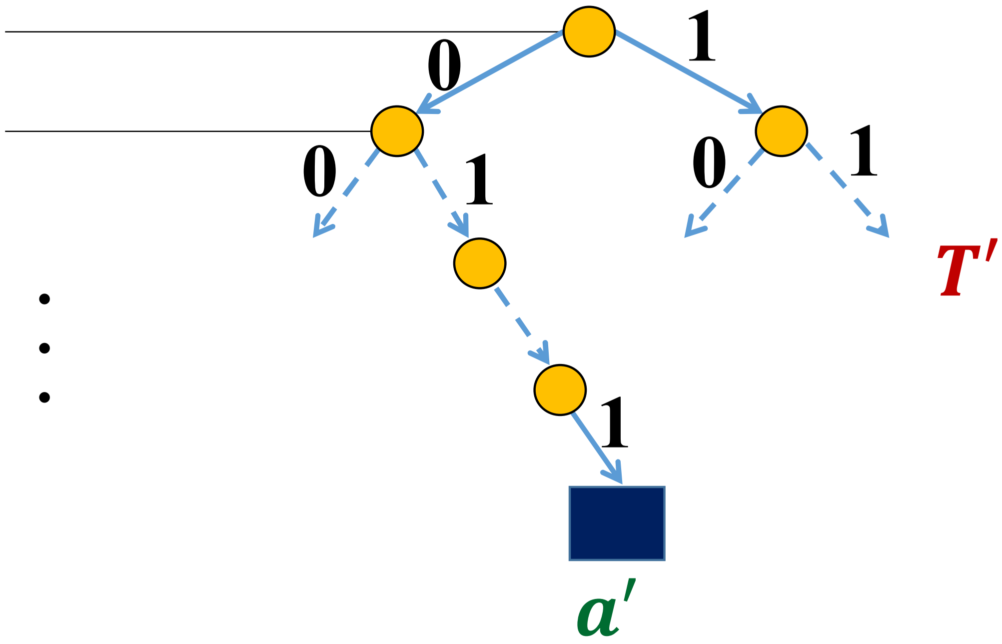
$$\boldsymbol{OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)} \ ?$$

Question 1: Can we derive a prefix coding for $\boldsymbol{A}$ from $\mathbf{OPT}(\boldsymbol{A'})$?

Question 2: Can we derive a prefix coding for $\boldsymbol{A'}$ from $\mathbf{OPT}(\boldsymbol{A})$?

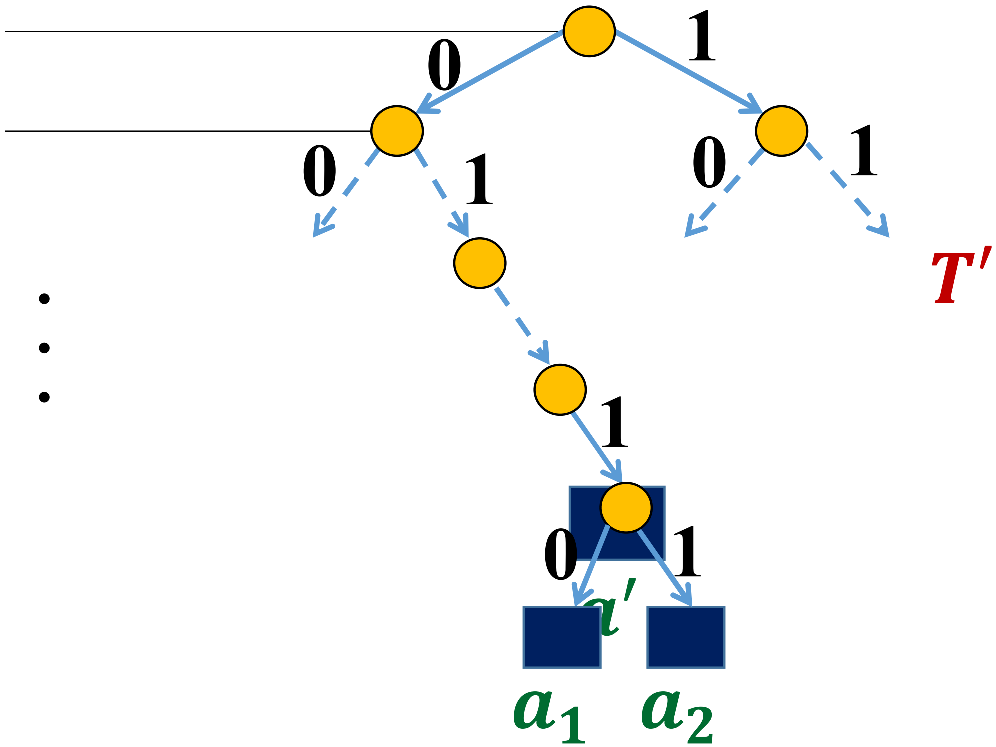
A prefix coding for $\boldsymbol{A}$ from $\mathbf{OPT}(\boldsymbol{A'})$

$\boldsymbol{T'}$: the binary tree corresponding to $\mathbf{OPT_{ABL}}(\boldsymbol{A'})$



A prefix coding for $\boldsymbol{A}$ from $\mathbf{OPT}(\boldsymbol{A'})$

$\boldsymbol{T'}$: the binary tree corresponding to $\mathbf{OPT_{ABL}}(\boldsymbol{A'})$



This gives a prefix coding for $\boldsymbol{A}$ with **ABL** = ??

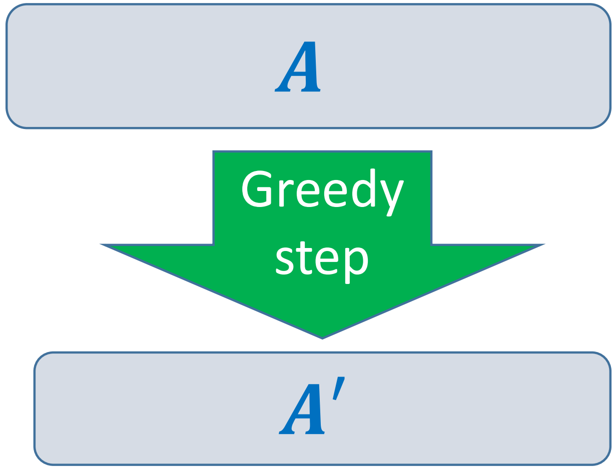
Question 2 at VA

Express **ABL** in terms of $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'})$, $\boldsymbol{f}(\boldsymbol{a_1})$ *and* $\boldsymbol{f}(\boldsymbol{a_2})$.

1. $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) + \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$
2. $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'})$
3. $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) + \max\{\boldsymbol{f}(\boldsymbol{a_1}), \boldsymbol{f}(\boldsymbol{a_2})\}$
4. $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) - \boldsymbol{f}(\boldsymbol{a_1}) - \boldsymbol{f}(\boldsymbol{a_2})$

To prove that a greedy strategy works

P: A given optimization problem



A: instance of size n of problem **P**

A': instance of size $< n$ of problem **P**

1. Try to establish a relation between **OPT**(A) and **OPT**(A');

2. Try to prove the relation formally by

- ❑ deriving a (not necessary optimal) solution of A from **OPT**(A')
- ❑ deriving a (not necessary optimal) solution of A' from **OPT**(A)

3. If you succeed, this would give you an algorithm.

Summary on Proof Techniques (for Greedy Algorithms)

- For Greedy Choice
  - Exchange argument

- For Optimal Substructure
  - Proof by contradiction; cut-and-paste argument
  - Constructive proof

Practice Problems (DP and Greedy)

- Tips to succeed for these two topics is…
- To solve as many problems as you can
  - Try solving exercises of textbooks (e.g., CLRS, CP4 ← ADS)
  - Look for more practice problems over Internet (Kattis, leetcode, old: UVa)

Acknowledgement

- The slides are modified from
  - The slides from Prof. Kevin Wayne
  - The slides from Prof. Surender Baswana
  - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - The slides from Prof. Arnab Bhattacharya and Prof. Wing-Kin Sung
  - The slides from Prof. Diptarka Chakraborty and Prof. Sanjay Jain