# *Analysis and Design of Algorithms*

CS3230

Week 9

## Greedy Algorithms

**Diptarka Chakraborty**

**Ken Sung**

# Dynamic Programming (Recap)

- Expressing the solution recursively

- Overall there are only polynomial number of subproblems

- But there is a huge overlap among the subproblems. So the recursive algorithm takes exponential time (solving same subproblem multiple times)

- So we compute the recursive solution iteratively in a bottom-up fashion (like in  case of Fibonacci numbers). This avoids wastage of computation and leads to an efficient implementation

# Question 4 (last lecture)

We have $n$ cents and need to get change in terms of denominations $d_1, d_2, \ldots, d_k$. Goal is to use the fewest total number of coins.

**Example:** If denominations are 25c, 10c, and 1c, then solution for $n =$30c should be 10c+10c+10c.

Let $M[j]$ be the fewest number of coins needed to change $j$ cents. Write a recursive formula for $M[j]$ in terms of $M[i]$ with $i < j$.

# Question 4: Solution

**Optimal substructure**: Suppose $M[j] = t$, meaning that

$$j = d_{i_1} + d_{i_2} + \cdots + d_{i_t}$$

for some $i_1, \ldots, i_t \in \{1, \ldots, k\}$. Then, if $j' = d_{i_1} + d_{i_2} + \cdots + d_{i_{t-1}}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

$$M[j] = \begin{cases} 1 + \min\limits_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$$

# Question 5 (last lecture)

Using the above, derive a DP algorithm to compute the minimum number of coins of denomination $d_1, \ldots, d_k$ needed to change $n$ cents.

# Question 5: Solution

NUM-COINTS-DP$(n, d)$:

**for** $j = 0, \dots, n$:

$\quad M[j] \leftarrow \infty$

$M[0] \leftarrow 0$

**for** $j = 1, \dots, n$:

$\quad$ **for** $i = 1, \dots, k$:

$\quad\quad$ **if** $(j - d_i \geq 0) \wedge (M[j - d_i] + 1 < M[j])$:

$\quad\quad\quad M[j] \leftarrow M[j - d_i] + 1$

**return** $M[n]$

Running time is $\Theta(nk)$.

# Today: Greedy Algorithms

A very general technique, like divide-and-conquer and dynamic programming

Technique is to recast the problem so that only one subproblem needs to be solved at each step. Beats divide-and-conquer and dynamic programming, **when it works**.

# Longest Increasing Subsequence

Applications in genome comparisons, …

# What is a subsequence?

Sequence $A : a_1, a_2, \ldots, a_n$

Can be stored in an array $A[1..n]$, $A[j] : a_j$ $\qquad A[1..k] : a_1, a_2, \ldots, a_k$

**Definition**: $C$ is said to be a **subsequence** of $A$ if we can obtain $C$ by removing 0 or more elements from $A$.

Example: $A : a, b, a, d, e, b, c, a$

$C : a, a, e, b, a$

A more formal definition:

$C$ is a **subsequence** of $A$ if there exists $k$ integers: $1 \le i_1 < \cdots < i_k \le n$ s.t.

for all $1 \le j \le k$ $\qquad C[j] = A[i_j]$

# Longest Increasing Subsequence (LIS)

**Problem:** Given a sequence of $n$ numbers find a longest increasing subsequence (**LIS**)

10, 3, 7, 2, 4, 6, 11, 8, 5, 9

# Question 1

Can you design an $O(n^2)$ time algorithm for the LIS problem using the LCS algorithm discussed in the last lecture?

# Longest Increasing Subsequence (LIS)

Given a sequence of n numbers

10, 3, 7, 2, 4, 6, 11, 8, 5, 9

$O(n \log n)$

1. Sort n numbers

   2, 3, 4, 5, 6, 7, 8, 9, 10, 11

2. Use dynamic programming to find longest common subsequence between sorted and the original sequence

$O(n^2)$

Can we do better?

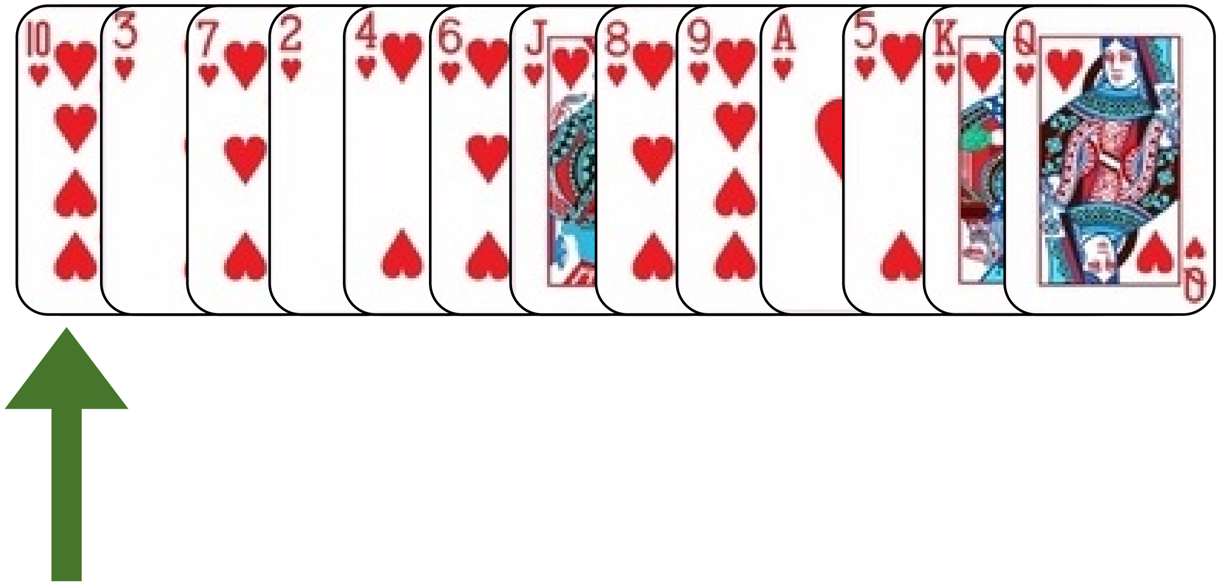
# Patience Solitaire

Given n cards deal into piles according to the following two rules

1. Not allowed to place a higher-valued card over a lower-valued card
2. Can form a new pile and place card

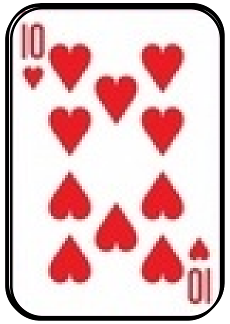
**Objective:** Form as few piles as possible.

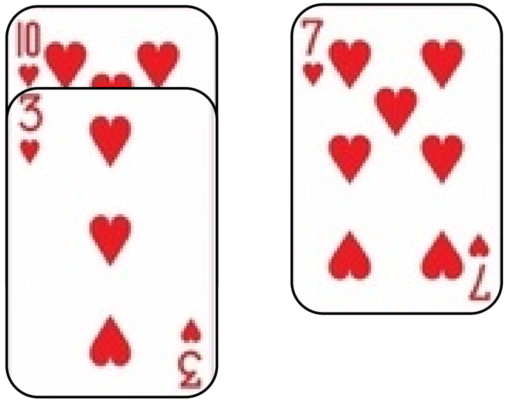
# Patience Solitaire




First card to deal

# Patience Solitaire



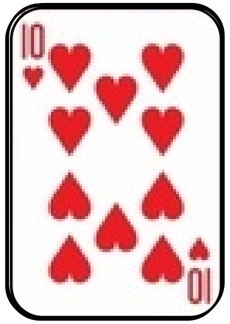


# Question 2

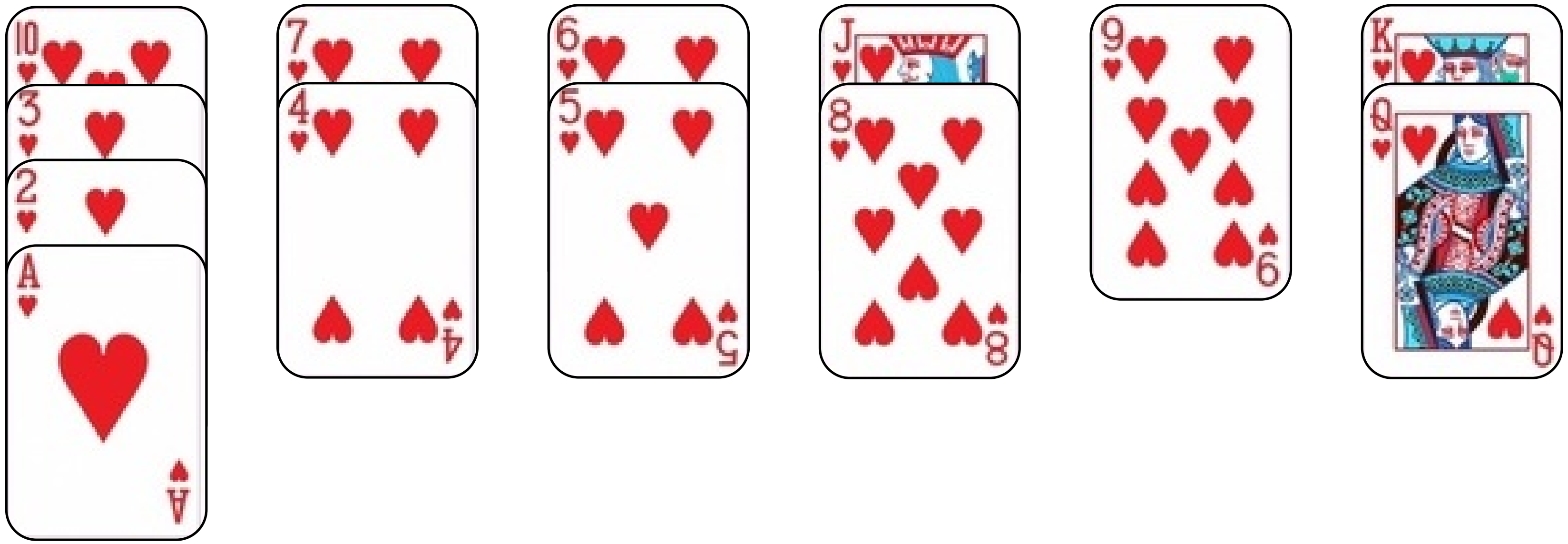
What strategy will you use while playing Patience Solitaire game so that you can always win?

1. Place card on the leftmost allowed pile; if not possible then create a new pile
2. Place card on the rightmost allowed pile; if not possible then create a new pile
3. Place card on any arbitrary allowed pile; if not possible then create a new pile
4. Since we cannot see the future cards we have to rely on luck to win. We could only ensure number of piles upto a constant times the minimum one.

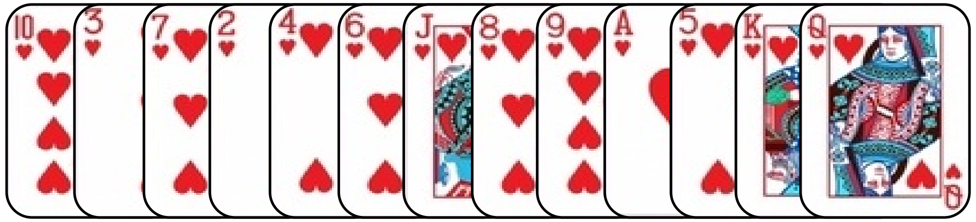
# Patience Solitaire



Greedy strategy: Place on the leftmost “allowed” pile; else create new pile

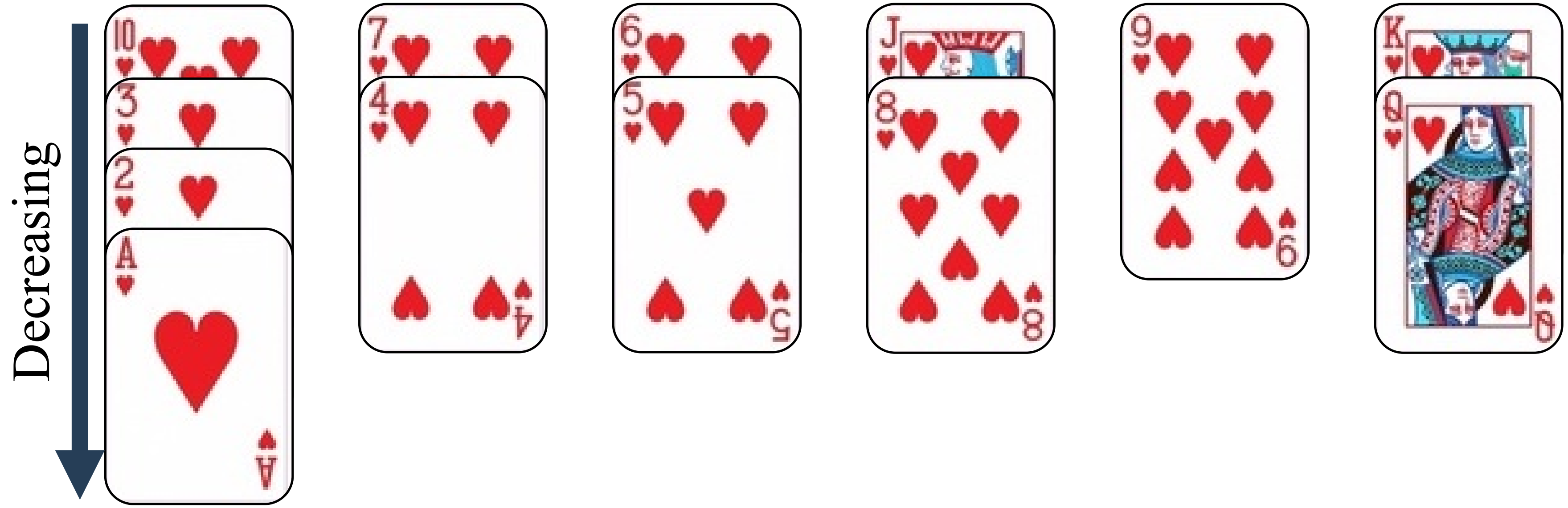


# Patience Solitaire



Observations:

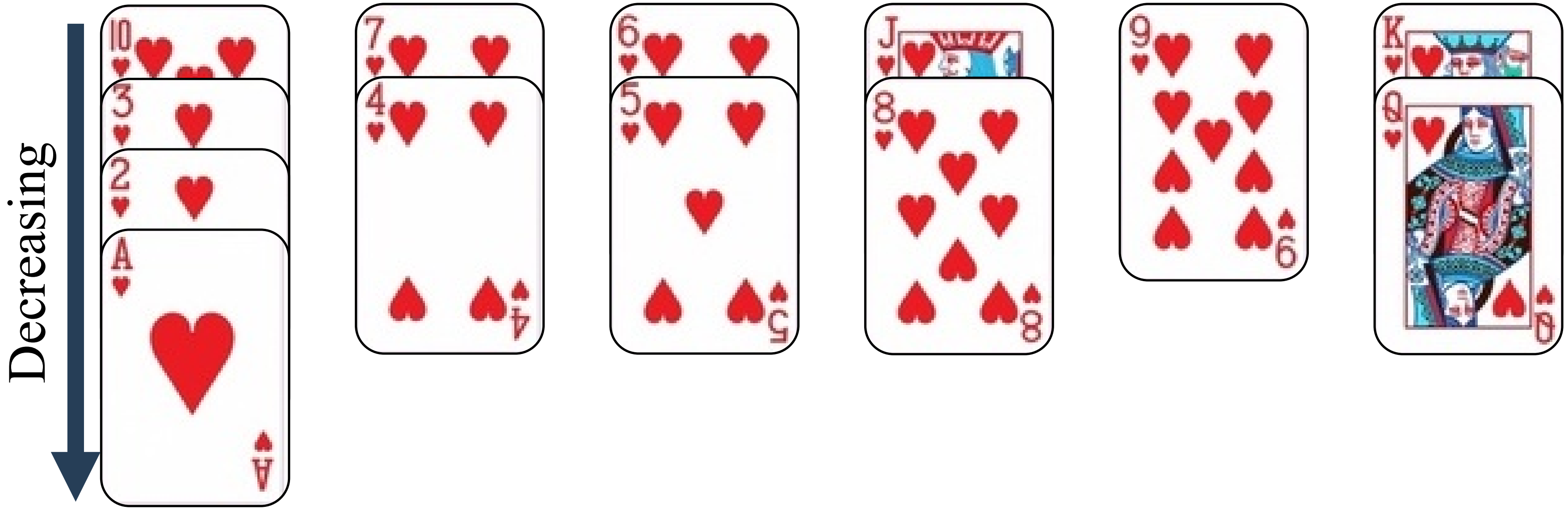
- Cards in each pile are in decreasing order
- Any **increasing subsequence** contains **at most one** card from each pile



# Patience Solitaire and LIS

Weak Duality:

Length of any increasing subsequence $\leq$ # of piles in any valid game



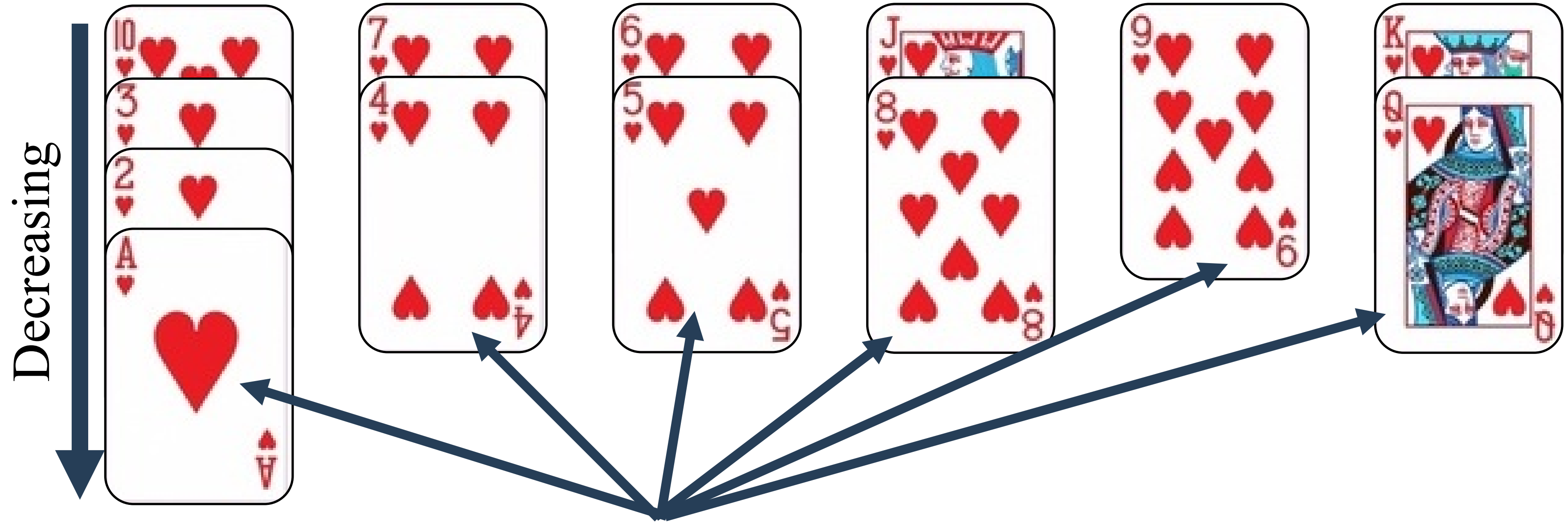
# Patience Solitaire and LIS

Length of LIS $\leq$ Min. number of piles

$\leq$ **# of piles in greedy strategy**

Why do not we take the top cards from each pile?

May not be a valid subsequence

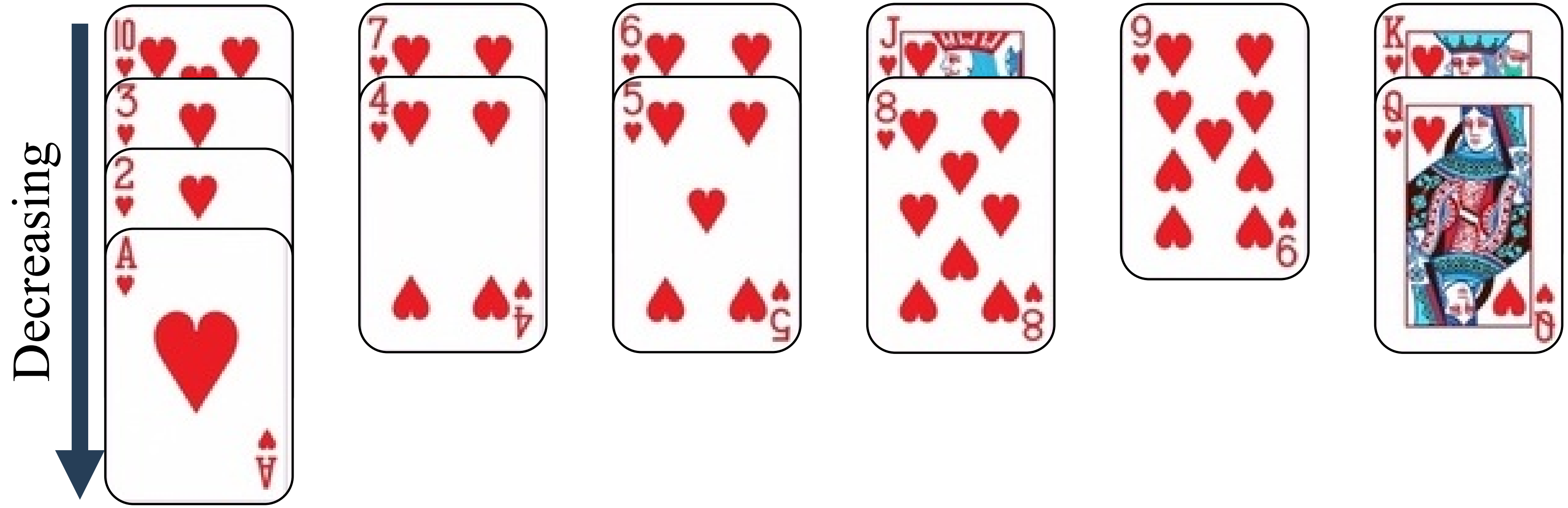


# Patience Solitaire and LIS

Length of LIS $\leq$ Min. number of piles

$\leq$ # of piles in greedy strategy

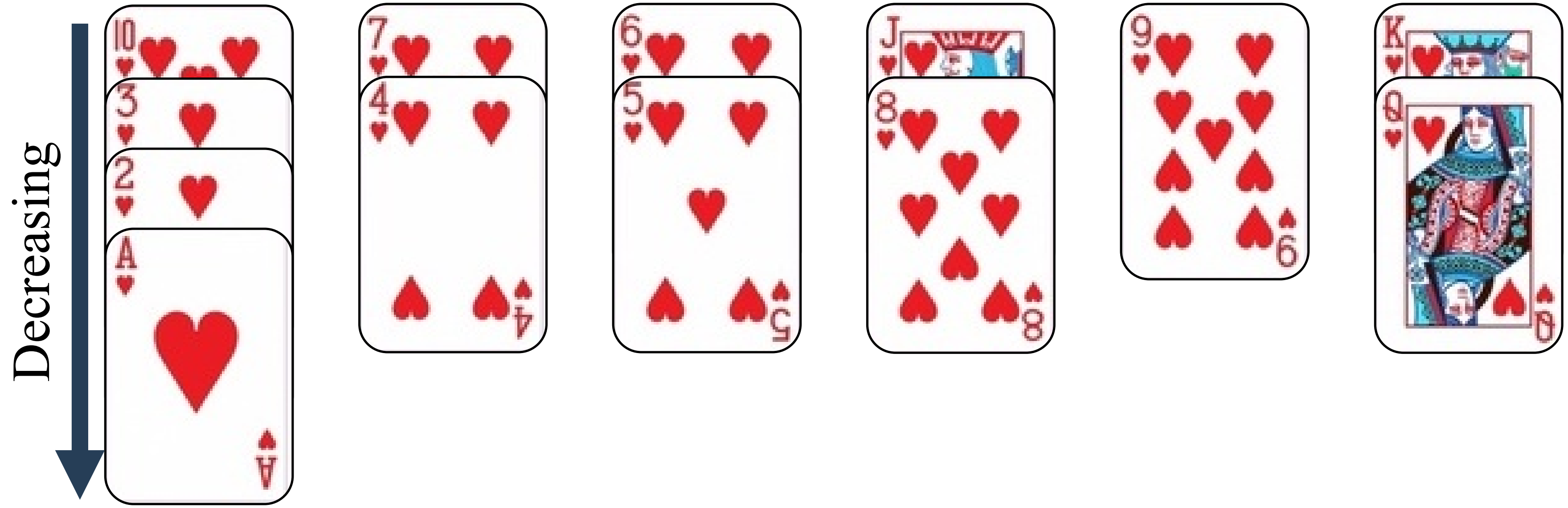
Can we take some card from each pile to form an increasing subsequence?



# Patience Solitaire and LIS

Strong Duality:

Length of LIS = Min. number of piles



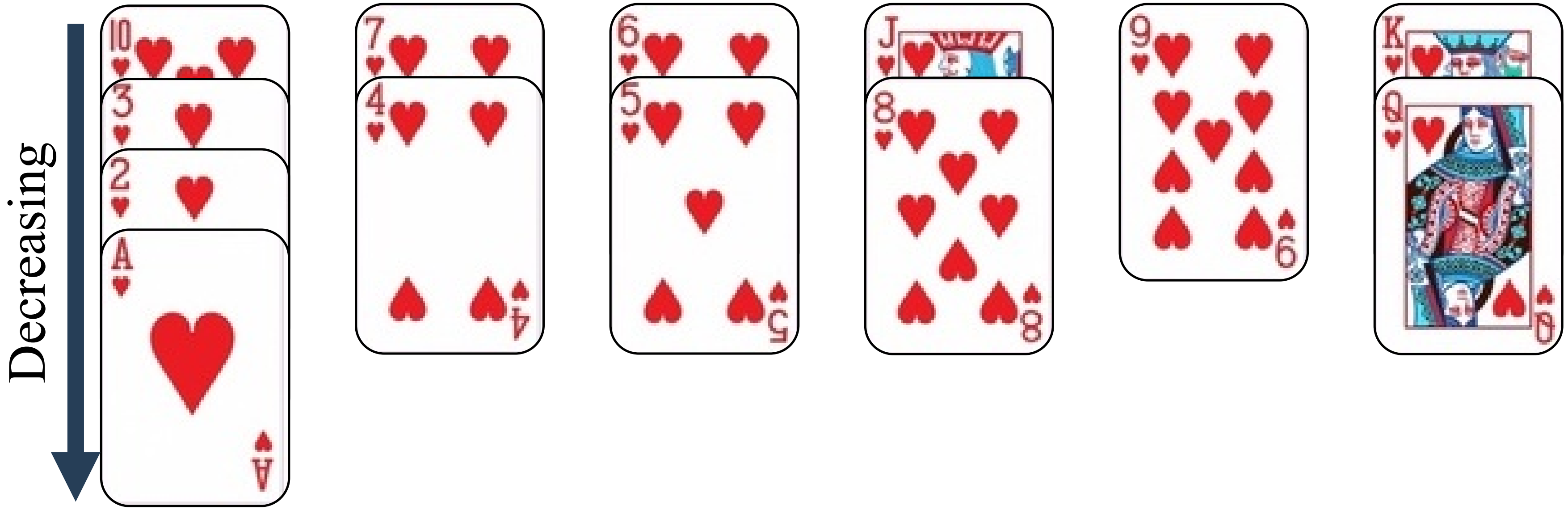
# Patience Solitaire and LIS

Length of LIS $\leq$ Min. number of piles

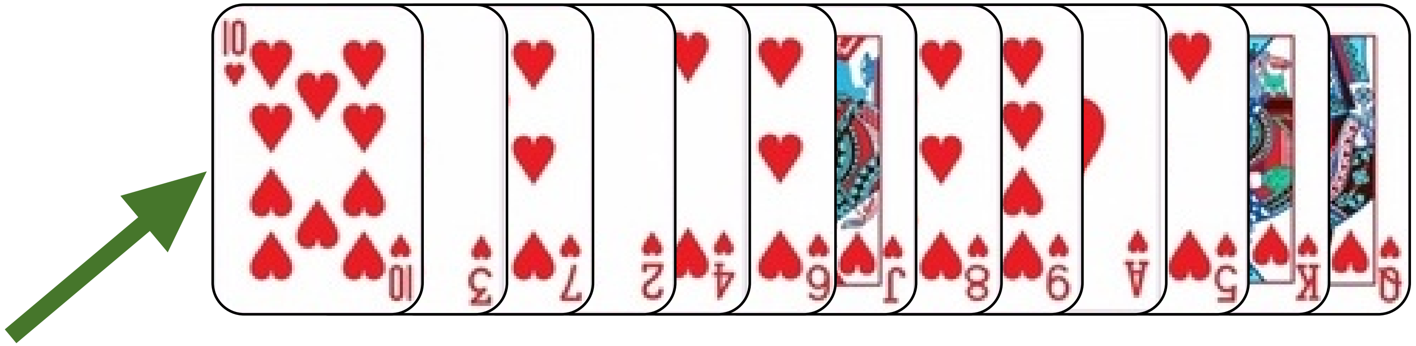
**$\leq$ # of piles in greedy strategy**

Can we take some card from each pile to form an increasing subsequence?

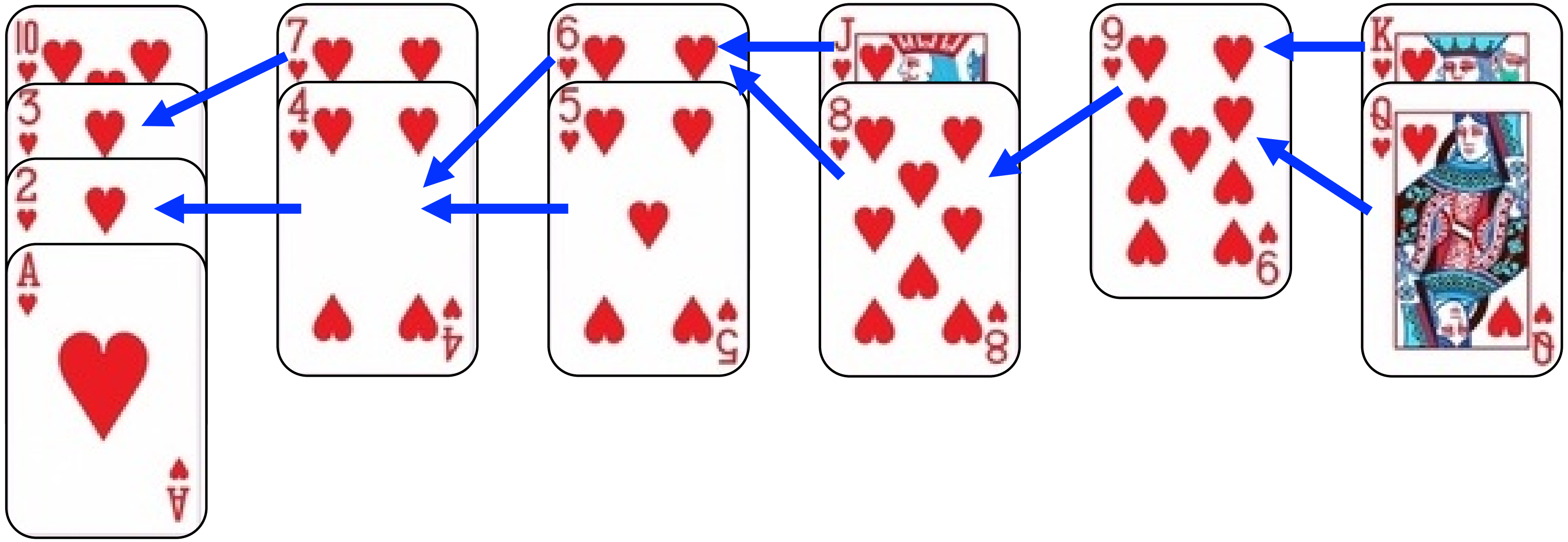
YES!!!
why?



# Patience Solitaire



Greedy strategy: Place on leftmost "allowed" pile; else create new pile



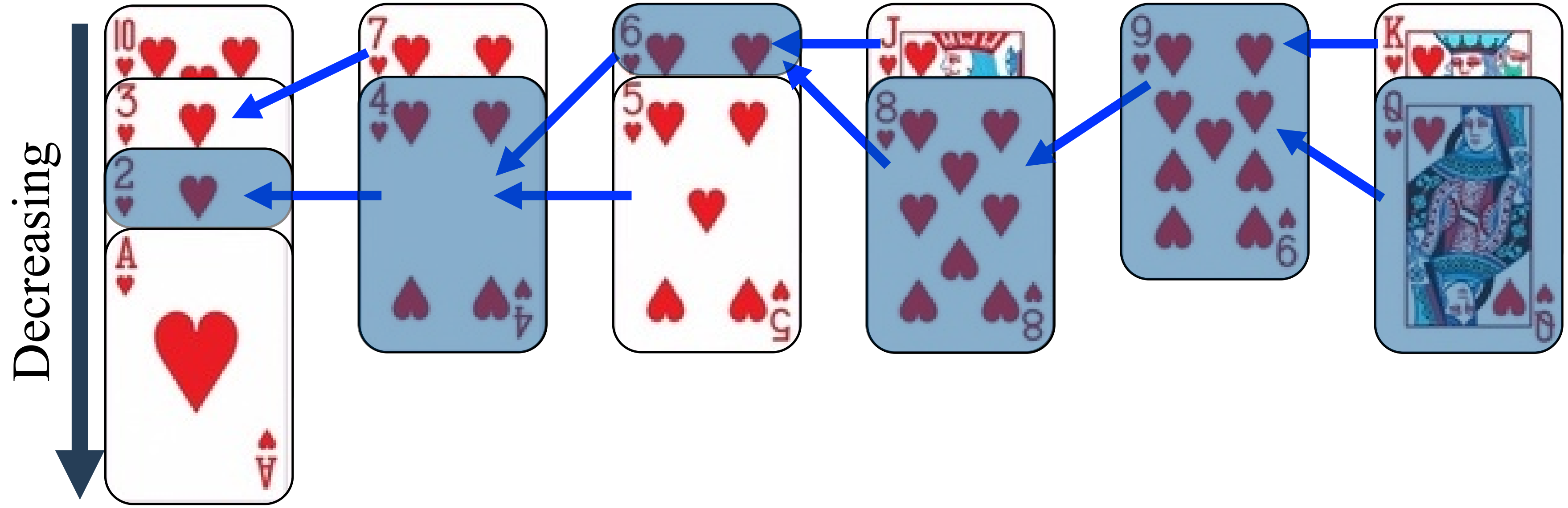
# Patience Solitaire and LIS

Length of LIS $\leq$ Min. number of piles

**$\leq$ # of piles in greedy strategy**

Can we take some card from each pile to form an increasing subsequence?

YES!!!
why?

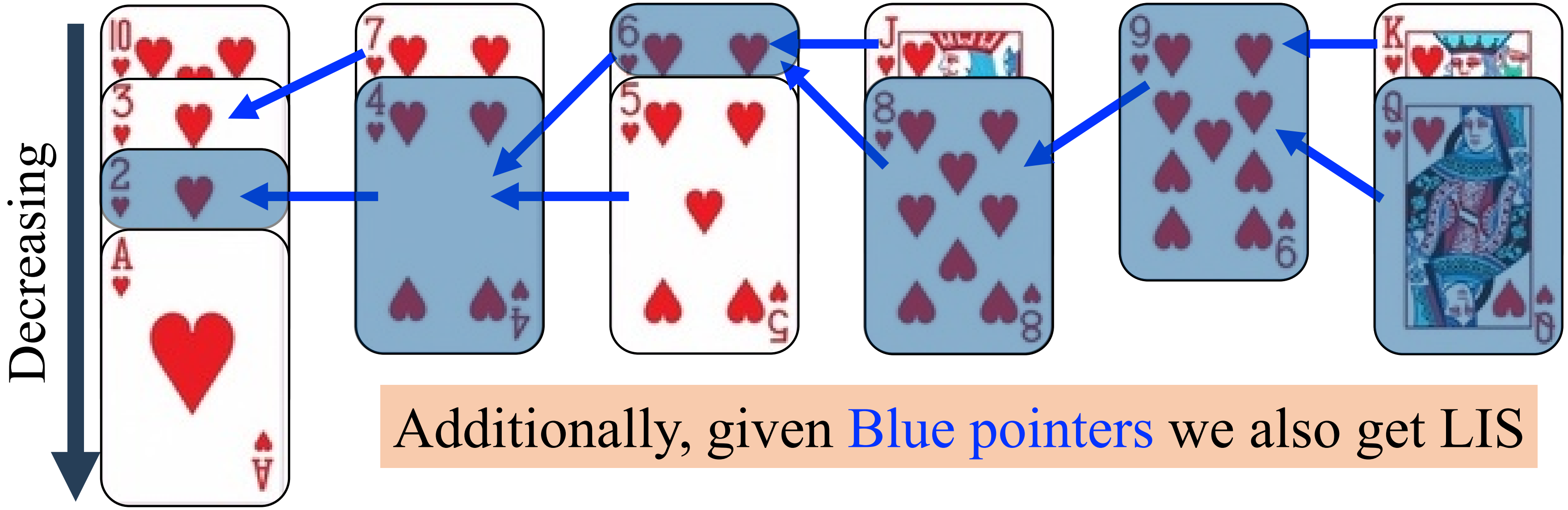


# Patience Solitaire and LIS

Strong Duality:

Found by our greedy strategy

Length of LIS = Min. number of piles




Additionally, given Blue pointers we also get LIS

# Running Time

Greedy strategy: Place on the leftmost “allowed” pile; else create new pile

Recall, top cards are in sorted order

At most **n** stacks

- Use stack to implement each pile
- How do you find the leftmost “allowed” pile?

Binary search

Time $= O(n \log n)$

# A few important points

- How do you use this greedy algorithm to sort?
- Each pile is sorted in decreasing order
- Simple Exercise: Use idea of merge sort to merge all the piles in total time $O(n \log n)$
- What is the best case in this sorting algorithm?
- If numbers are $1,2, \ldots, n$ then the above algorithm can be implemented in time $O(n \log \log n)$

Lets leave it for advanced algorithm course !!!

# Paradigm for greedy algorithms

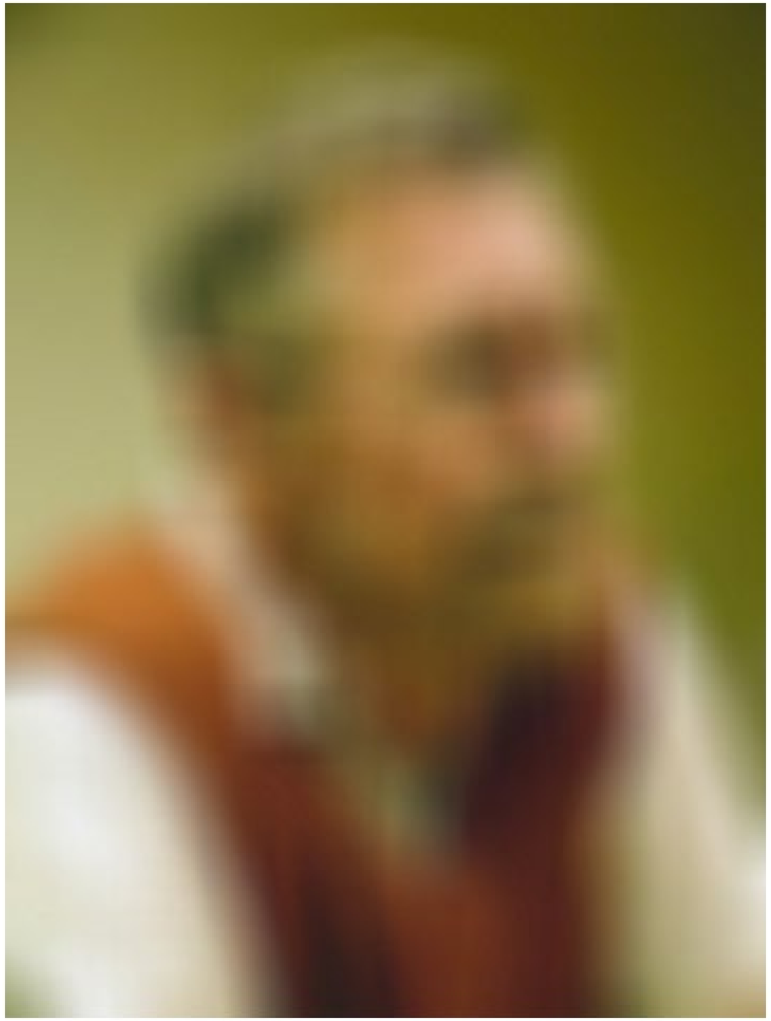
1. Cast the problem where we have to make a choice and are left with one subproblem to solve.

2. Prove 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 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.

# Question 3

Who is the Master of Algorithms pictured below?



- Edsger Dijkstra
- Robert Prim
- Joseph Kruskal
- Jack Edmonds

# Question 3

**Edsger Dijkstra**

Turing Award winner

Known for Dijkstra's algorithm (greedy algorithm!), structured programming (GOTO statement considered harmful), semaphores, mutex, deadlock (Dining Philosophers problem), ...

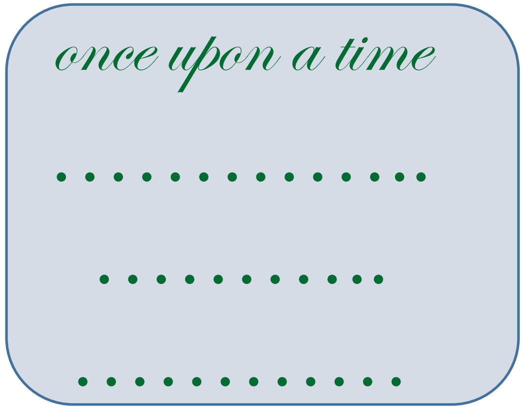
# Huffman Code

Applications in data compression, …

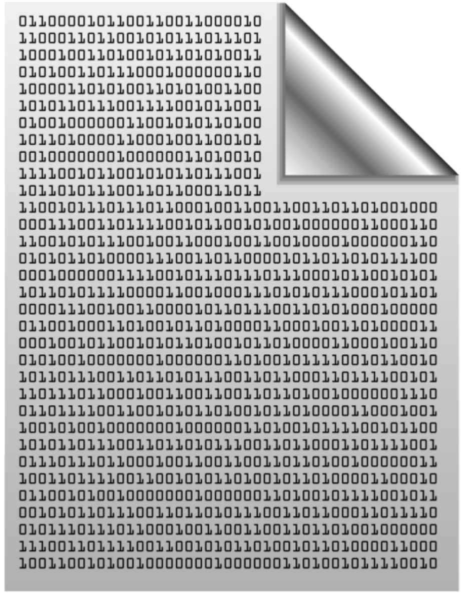
# Binary coding

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

**A text File**: a sequence of alphabets




**A text file F**



**Binary coding of F**

**Question**: How many bits needed to encode a text file with $\boldsymbol{m}$ characters?

**Answer**: $\boldsymbol{m}\ \lceil \log_2 \boldsymbol{n} \rceil$ bits.

# Fixed length coding

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**

**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 ☺

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

# Fixed length coding

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

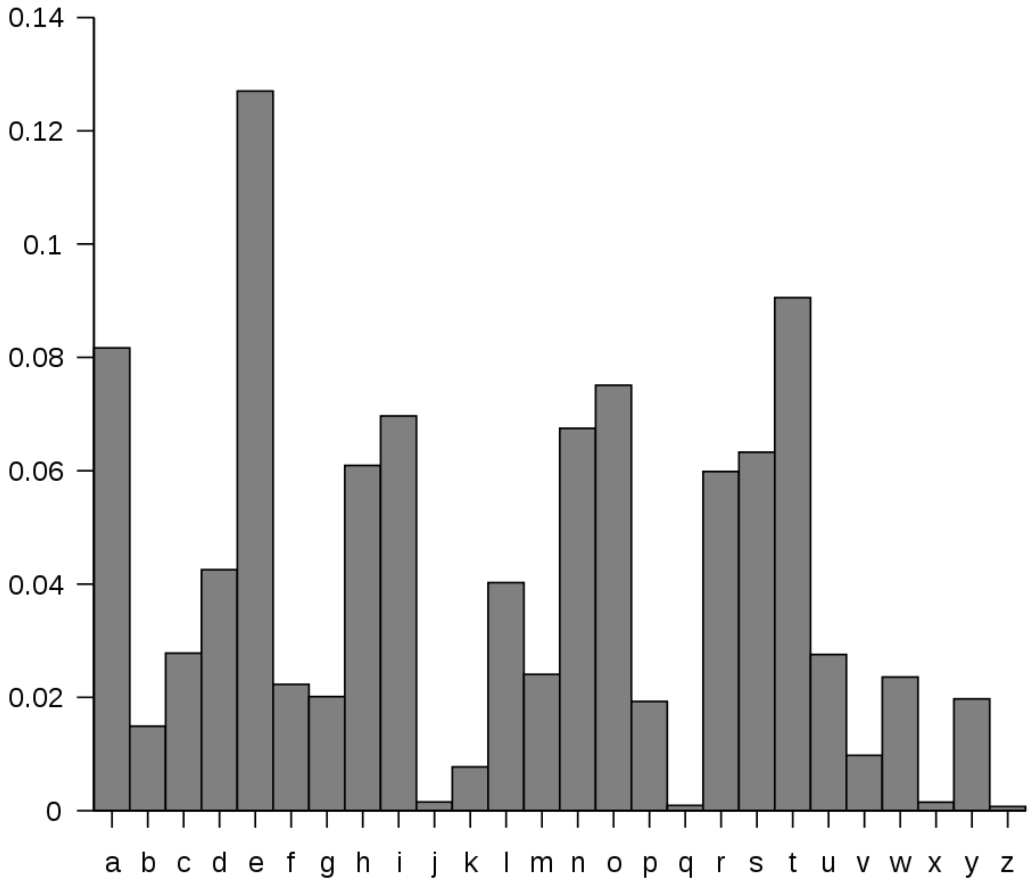
**Question**: Can we use fewer bits to store alphabet set $\boldsymbol{A}$ ?

**Answer**: No.

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

**Answer**: Yes

# Huge variation in the frequency of alphabets in a text




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

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

**Answer**:

**More frequent** alphabets ← coding with **shorter bit string**

**Less frequent** alphabets ← coding with **longer bit string**

# Variable length encoding

| Alphabets | Frequency $f$ | Encoding $\gamma$ |
|---|---|---|
| $a$ | 0.45 | 0 |
| $b$ | 0.18 | 10 |
| $c$ | 0.15 | 110 |
| $d$ | 0.12 | 101 |
| $e$ | 0.10 | 111 |

**Average bit length** per symbol using $\gamma$:

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

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

$$= \mathbf{1.92}$$

There is a serious problem with the $\gamma$ encoding. Can you see?

# Variable length encoding

**Can you fix it?**

| ets | Frequency $f$ | Encoding $\gamma$ |
|---|---|---|
| $a$ | 0.45 | 0 |
| $b$ | 0.18 | 10 |
| $c$ | 0.15 | 110 |
| $d$ | 0.12 | 101 |
| $e$ | 0.10 | 111 |

**Average bit length** per symbol using $\gamma$:

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

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

$$= \mathbf{1.92}$$

**Question**: How will you decode 01010111 ?

**Answer**: $\boldsymbol{abbe}$ or $\boldsymbol{adae}$ ☹

**Question**: What is the source of this ambiguity ?

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

# Variable length Coding

| Alphabets | Frequency $f$ | Encoding $\gamma$ |
|---|---|---|
| $a$ | 0.45 | 0 |
| $b$ | 0.18 | 100 |
| $c$ | 0.15 | 110 |
| $d$ | 0.12 | 101 |
| $e$ | 0.10 | 111 |

**Average bit length** per symbol using $\gamma$:

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

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

$$= \mathbf{2.1}$$

# Prefix Coding

**Definition**:

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

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

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

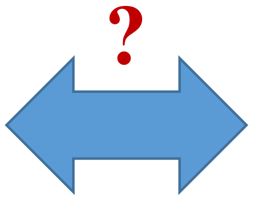
- $\boldsymbol{\gamma}$ is prefix coding
- $\mathbf{ABL}(\boldsymbol{\gamma})$ is **minimum**.

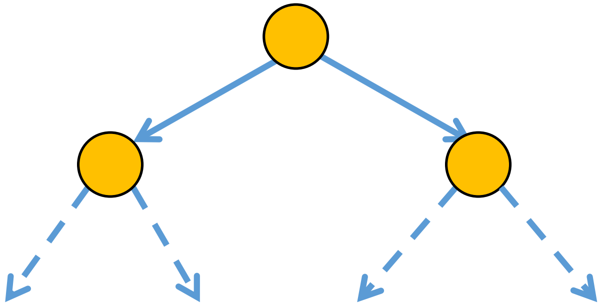
# The challenge of the problem

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

# The novel idea of Huffman

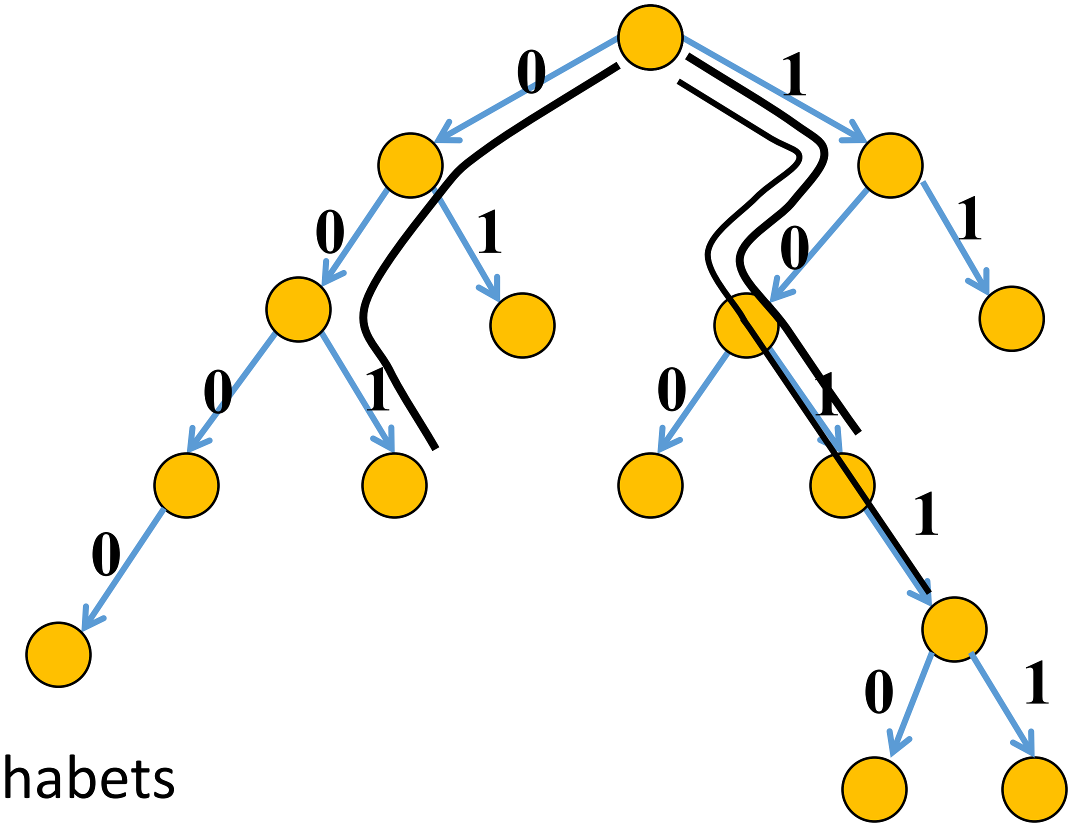
Binary coding



Binary tree

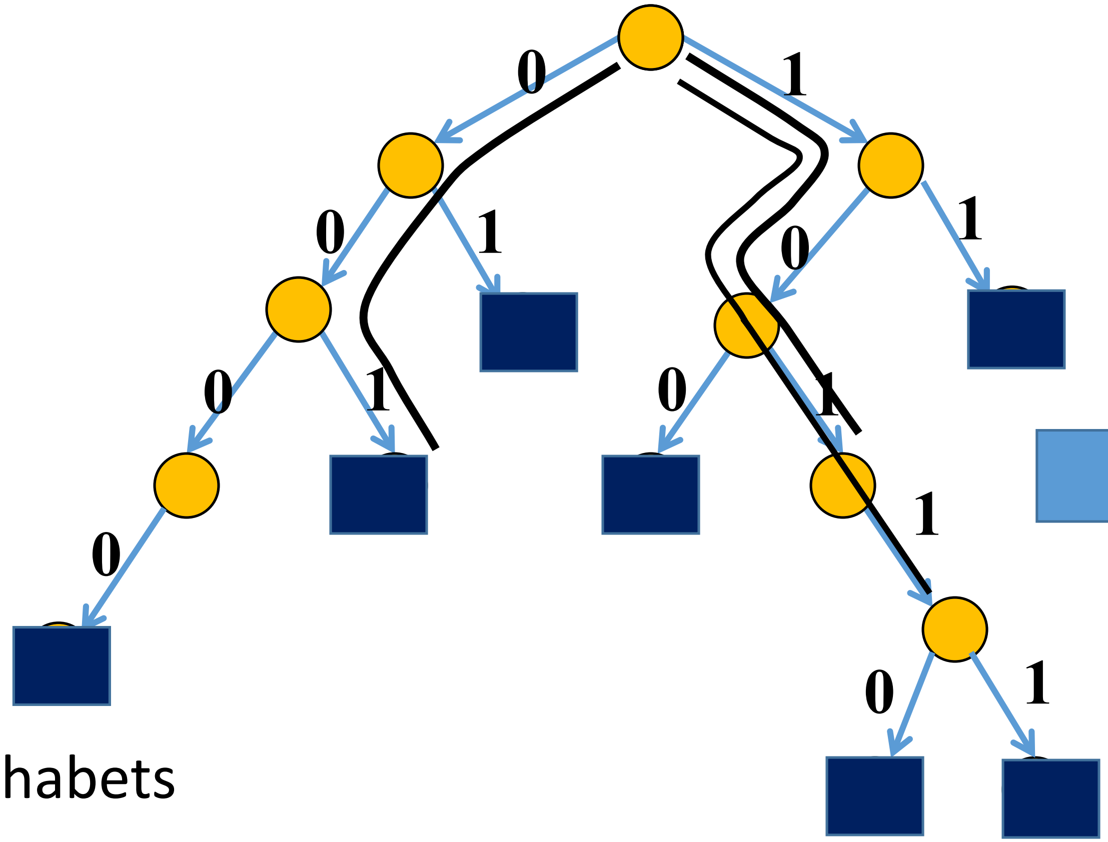
# A labeled binary tree




**Leaf** nodes → alphabets

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

# A labeled binary tree




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

**Leaf** nodes → alphabets

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

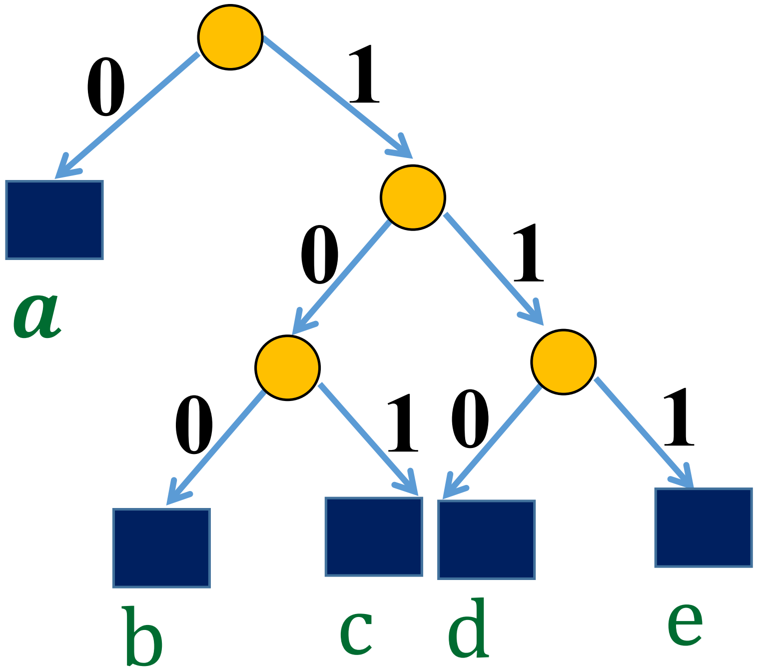
# Variable length Coding

| Alphabets | Frequency $f$ | Encoding $\gamma$ |
|---|---|---|
| $a$ | 0.45 | 0 |
| $b$ | 0.18 | 100 |
| $c$ | 0.15 | 110 |
| $d$ | 0.12 | 101 |
| $e$ | 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 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.

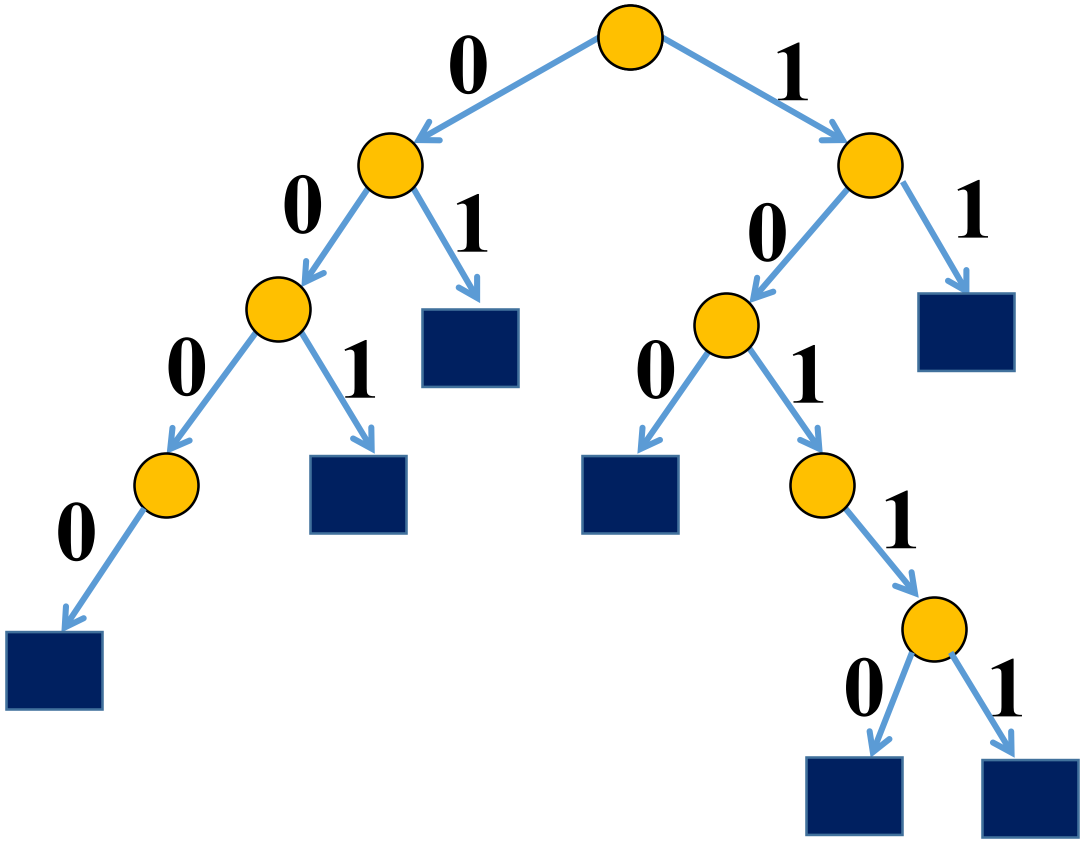
**Question:** Can you express **Average bit length** of $\boldsymbol{\gamma}$ in terms of its binary tree $\boldsymbol{T}$ ?

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

$$= \sum_{x \in \boldsymbol{A}} f(x).|\mathbf{depth}_{\mathbf{T}}(x)|$$

# Finding the labeled binary tree for the optimal prefix codes

# Is the following prefix coding optimal ?




NO

# Observations on the binary tree of the optimal prefix code

**Lemma**:

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 the optimal binary tree.

Let $\boldsymbol{a_1}$, $\boldsymbol{a_2}$,…, $\boldsymbol{a_n}$ be the alphabets of $\boldsymbol{A}$ in <u>non-decreasing</u> order of their frequencies.

# Observations on the binary tree of the 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 ?

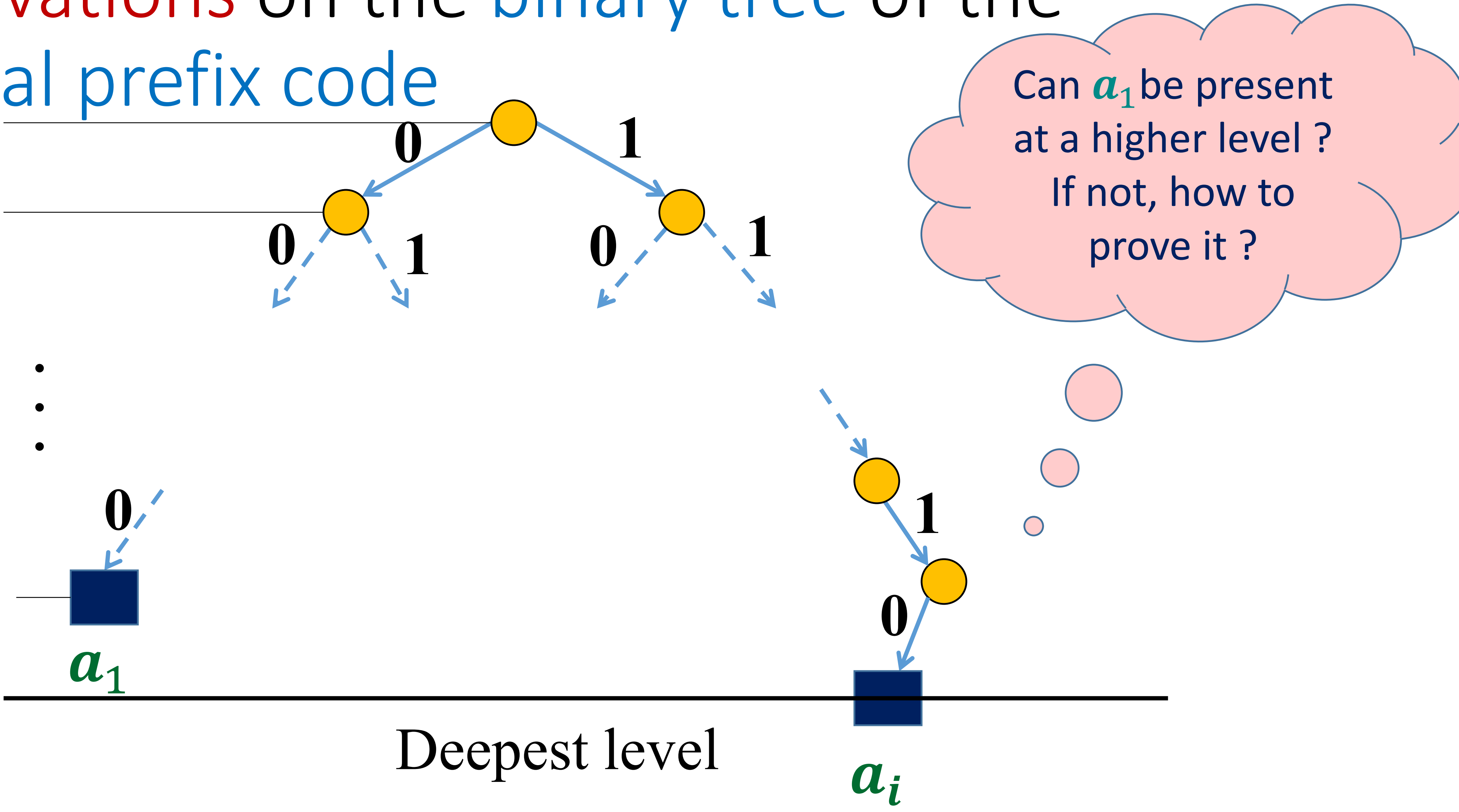
- We shall now make some simple observations about the structure of the binary tree corresponding to the optimal prefix codes.

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

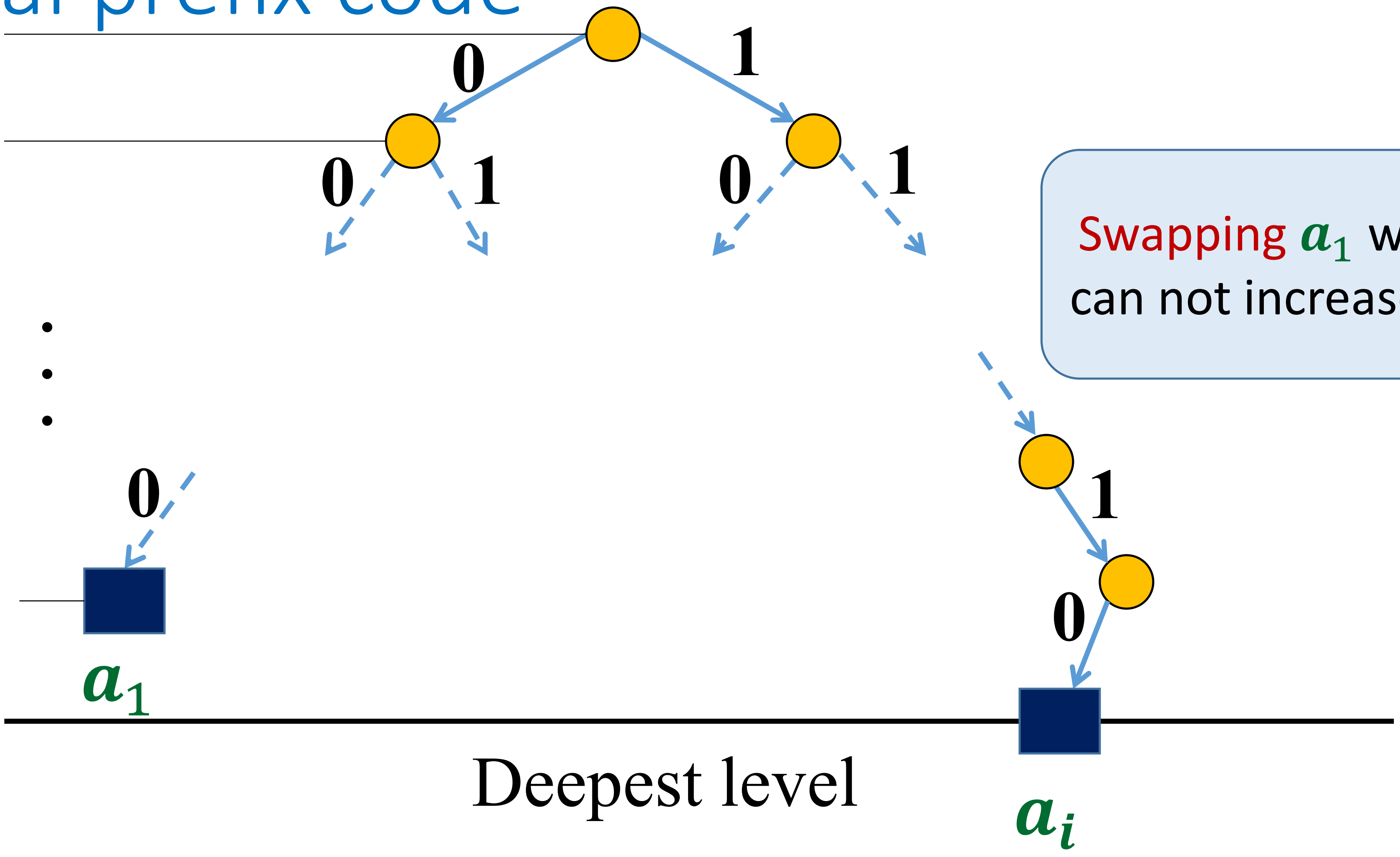
- Nevertheless, these observations will play a crucial role in the design of a binary tree with optimal prefix code for given $\boldsymbol{A}$.

Please pay full attention on the next few slides.

# Observations on the binary tree of the optimal prefix code

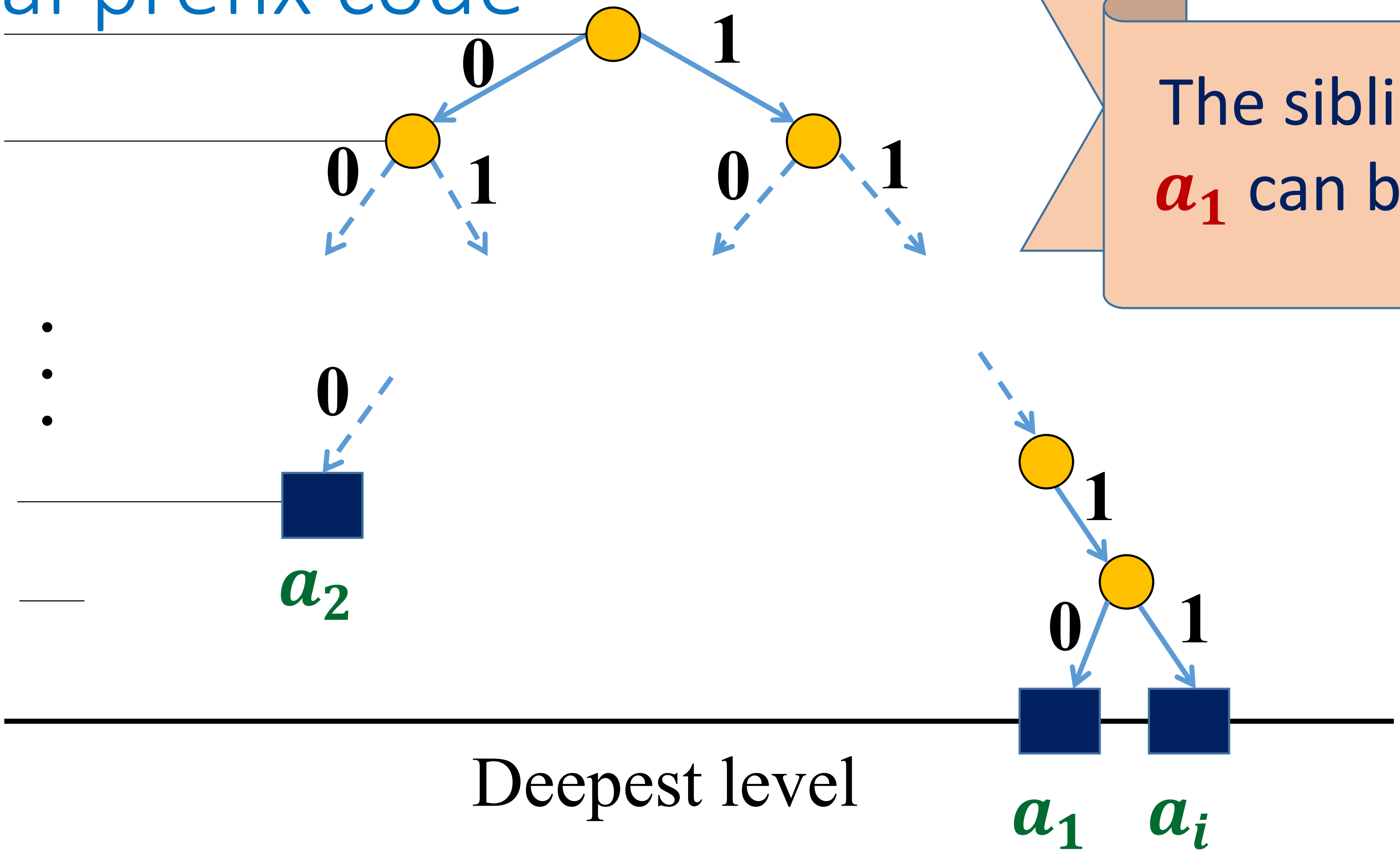


# Observations on the binary tree of the optimal prefix code




Swapping $a_1$ with $a_i$ can not increase **ABL**.

# Observations on the binary tree of the optimal prefix code




The sibling of $a_1$ can be $a_2$.

# 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.

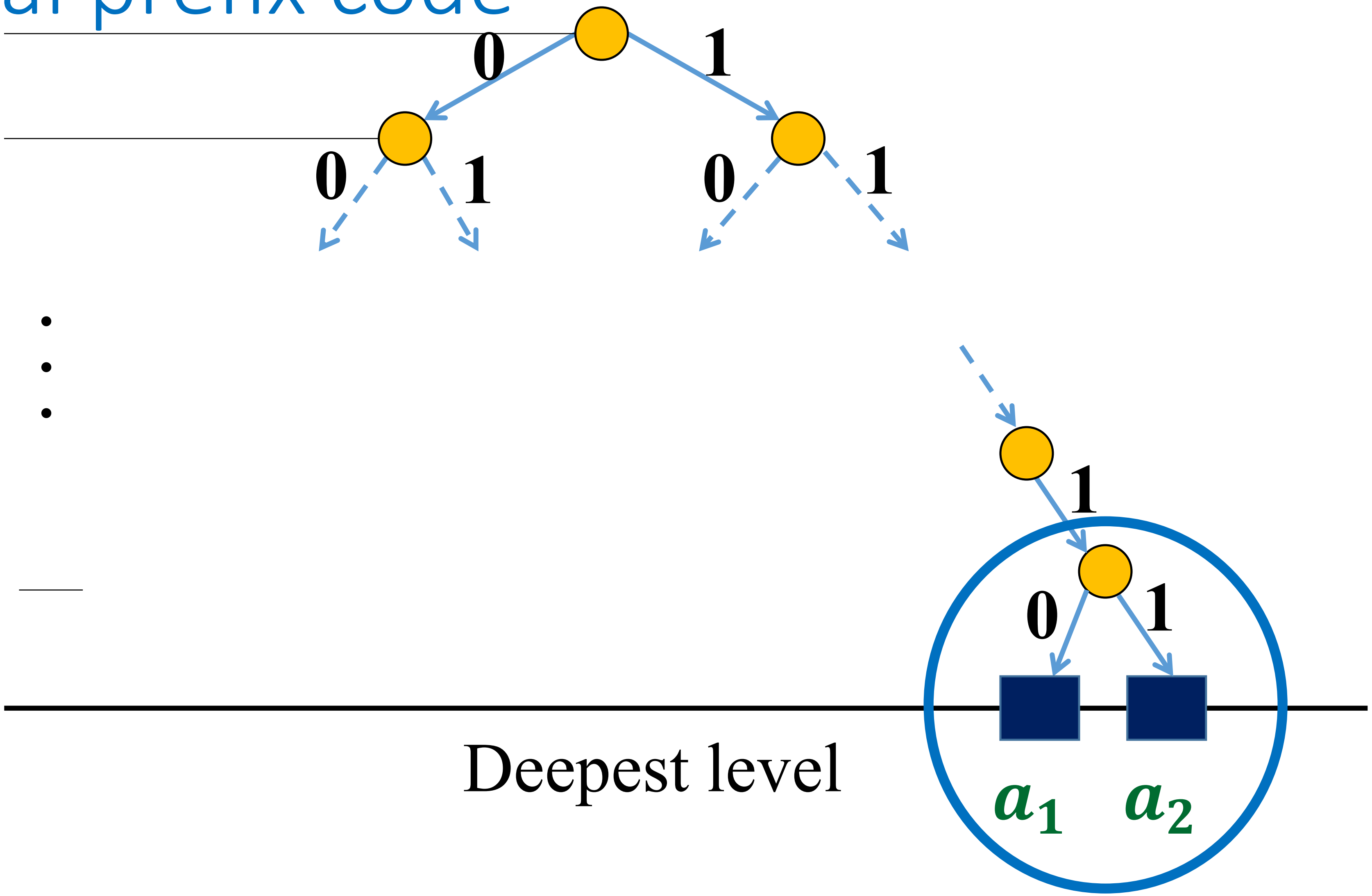
**Important note**: It is inaccurate to claim that “In every optimal prefix coding, $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ appear as siblings in the labeled binary string.”

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

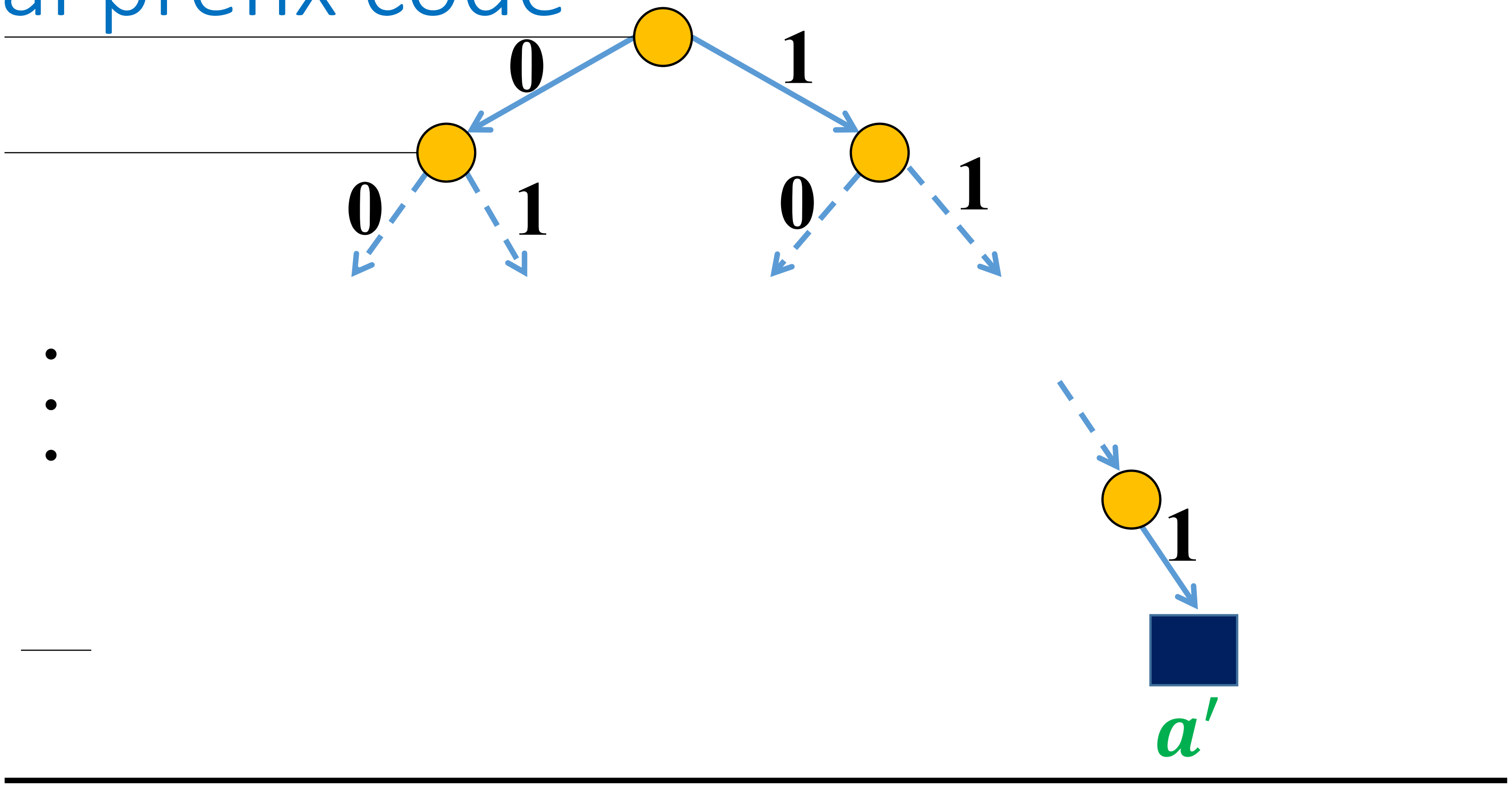
➔ We just need to focus on that binary tree of optimal prefix coding in which $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ appear as siblings.

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

# Observations on the binary tree of the optimal prefix code



# Observations on the binary tree of the optimal prefix code




Deepest level

# Key Idea to design an algorithm

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

$\boldsymbol{A'} = \boldsymbol{a_3}, \ldots, \boldsymbol{a'}, \ldots, \boldsymbol{a_n}$ be $\boldsymbol{n-1}$ alphabets in increasing 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 :** The optimal prefix code of $\boldsymbol{A'}$ ➔ optimal prefix code of $\boldsymbol{A}$

# Key Idea to design an algorithm

$A = a_1, a_2, \ldots, a_n$ be $n$ alphabets in increasing order of frequencies

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

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

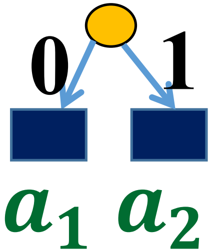
**Answer**: $\mathbf{OPT}_{\mathbf{ABL}}(A) = \mathbf{OPT}_{\mathbf{ABL}}(A') + f(a_1) + f(a_2)$

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

# The algorithm based on

$$\mathrm{OPT}_{\mathrm{ABL}}(A) = \mathrm{OPT}_{\mathrm{ABL}}(A') + f(a_1) + f(a_2)$$

**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 $A$;

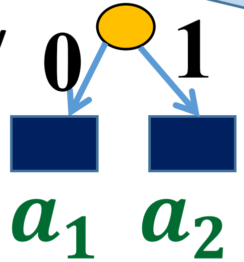
**Create** a new alphabet $a'$;

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

**Insert** $a'$ into $A$;

$T \leftarrow$ **OPT**$(A)$;

**Replace** node 

in $T$ by 

;

return $T$;

}}

**Sort** the alphabets according to frequencies
Takes $O(n \log n)$ time

Do binary search to update the sorted list in $O(\log n)$ time

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

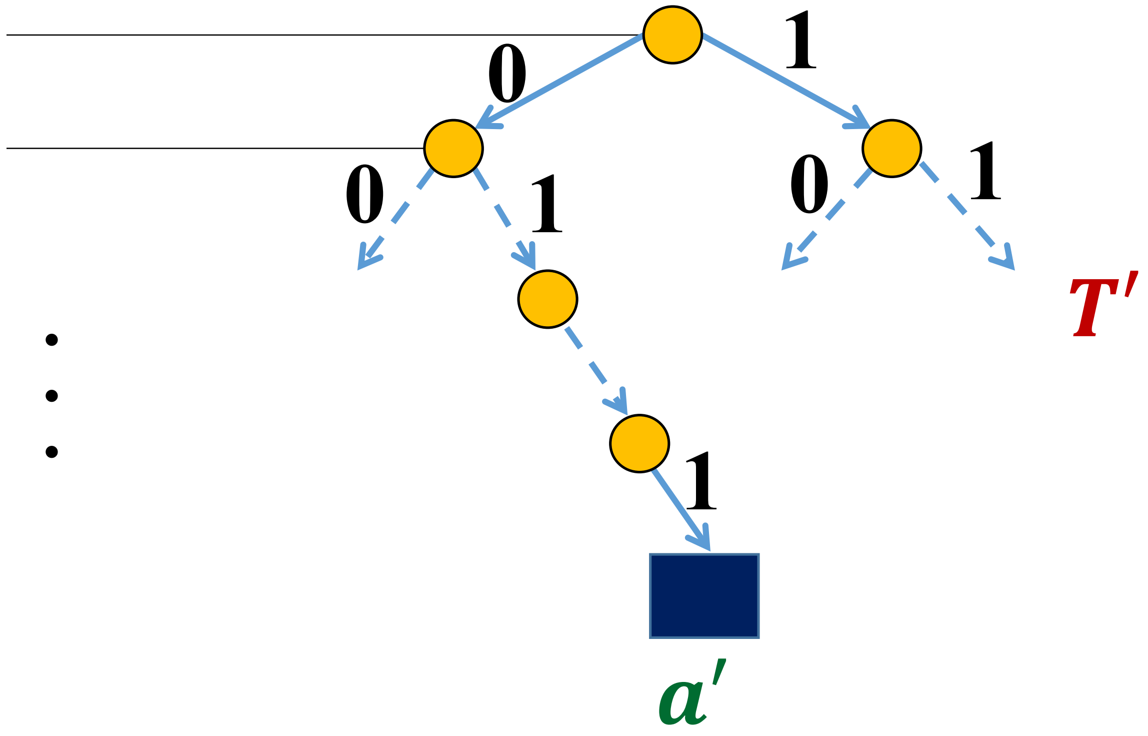
# How to prove $\mathrm{OPT}_{\mathrm{ABL}}(A) = \mathrm{OPT}_{\mathrm{ABL}}(A') + f(a_1) + f(a_2)$ ?

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

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

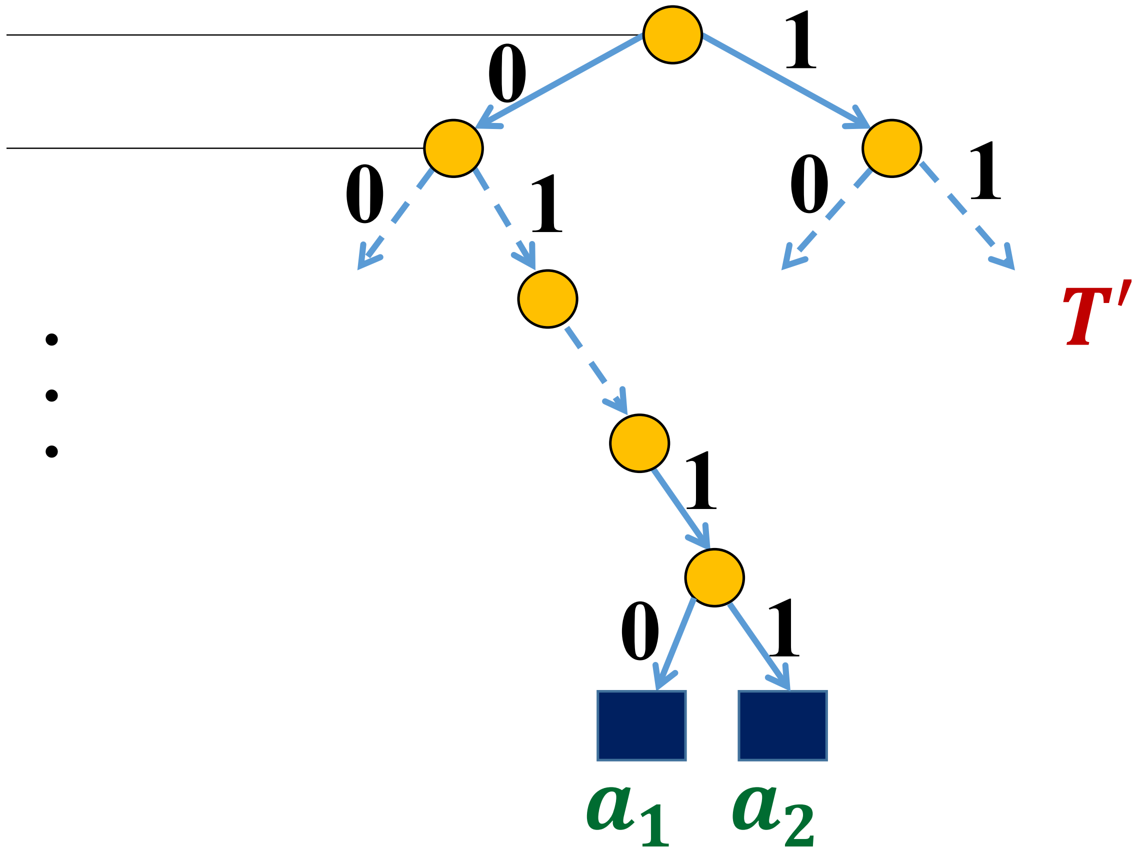
# A prefix coding for $\boldsymbol{A}$ from $\mathbf{OPT}(\boldsymbol{A'})$

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



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

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



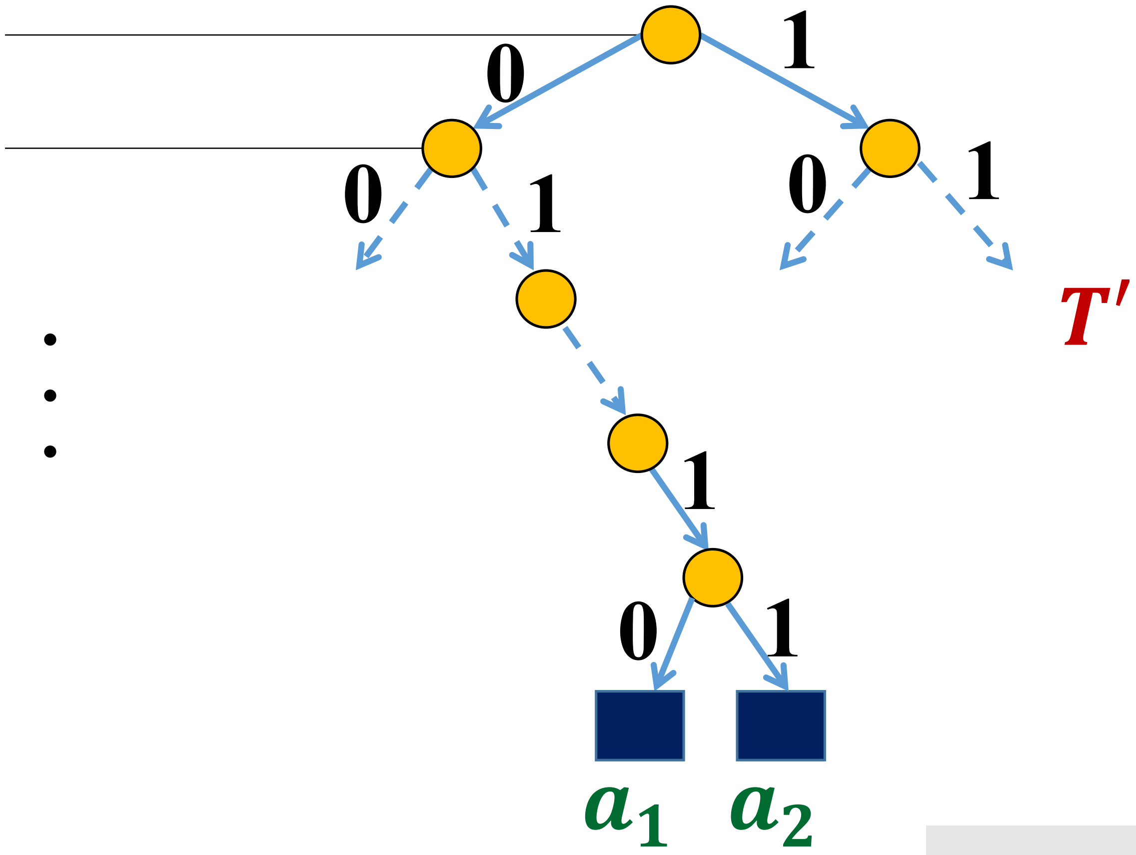

This gives a prefix coding for $\boldsymbol{A}$ with $\mathbf{ABL}$ = ??

# Question 4

Express ABL in terms of $\mathrm{OPT}_{\mathrm{ABL}}(\boldsymbol{A}')$, $\boldsymbol{f}(\boldsymbol{a}_1)$ $and$ $\boldsymbol{f}(\boldsymbol{a}_2)$.

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

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

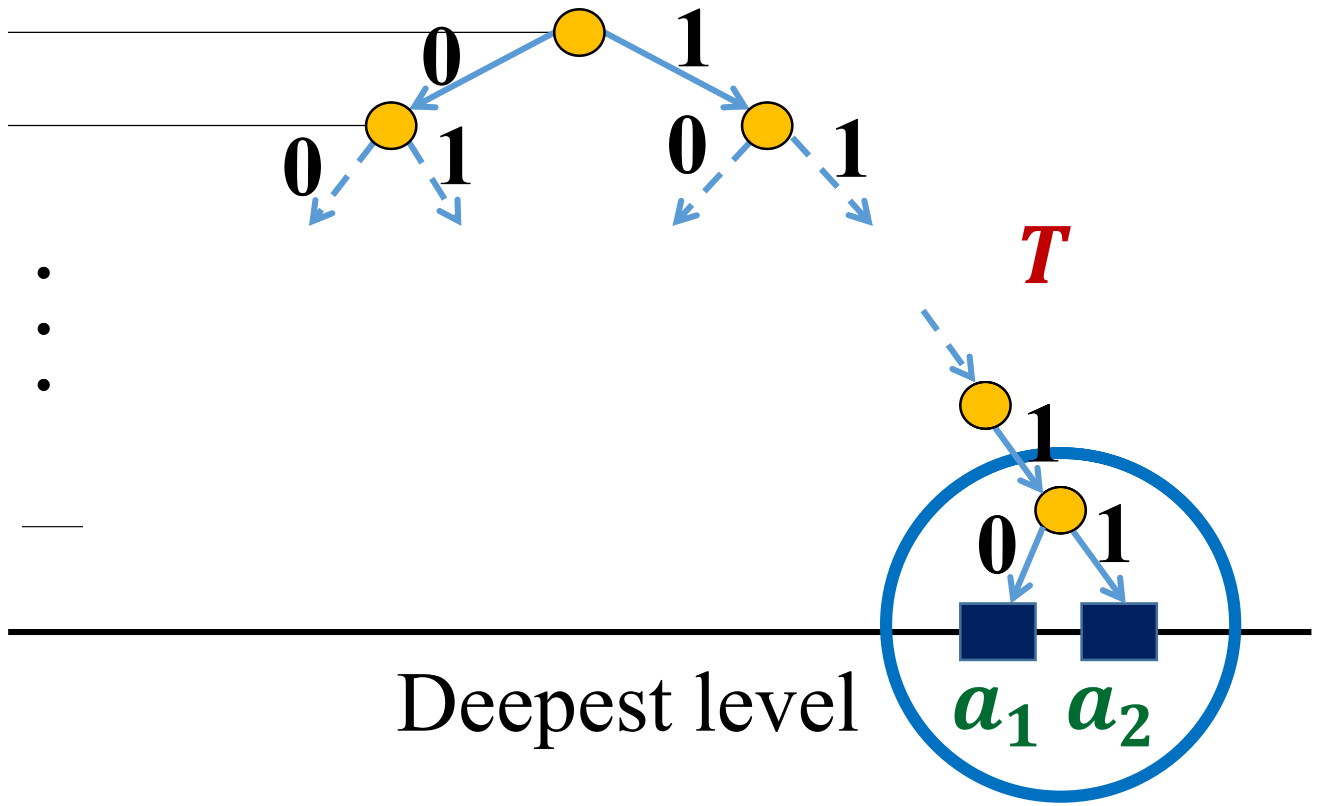



This gives a prefix coding for $\boldsymbol{A}$ with $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) + \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$

$\Rightarrow \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) \le \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A'}) + \boldsymbol{f}(\boldsymbol{a_1}) + \boldsymbol{f}(\boldsymbol{a_2})$

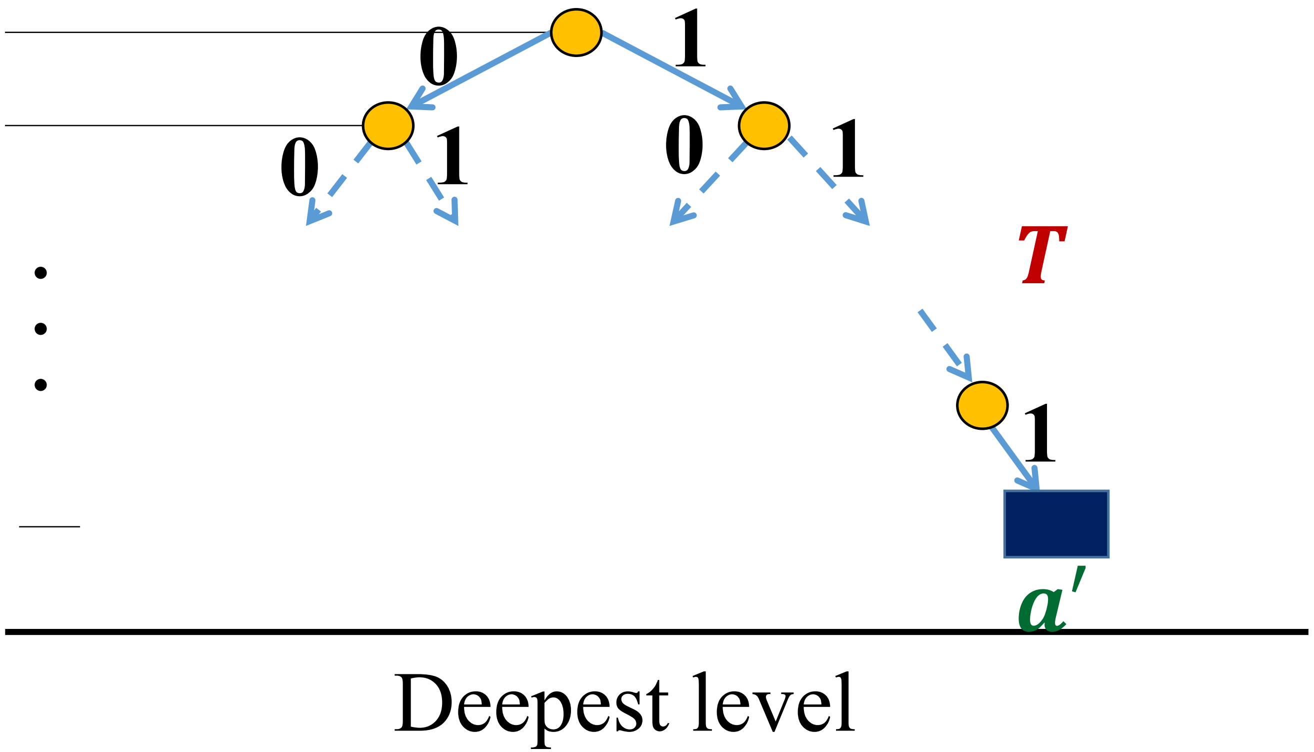
# A prefix coding for $\boldsymbol{A'}$ from $\mathbf{OPT}(\boldsymbol{A})$

$\boldsymbol{T}$: the binary tree corresponding to $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A})$



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

$T$: the binary tree corresponding to $\mathbf{OPT}_{\mathbf{ABL}}(A)$




This gives a prefix coding for $A'$ with $\mathbf{ABL} = \mathbf{OPT}_{\mathbf{ABL}}(A) - f(a_1) - f(a_2)$

➔ $\mathbf{OPT}_{\mathbf{ABL}}(A') \le \mathbf{OPT}_{\mathbf{ABL}}(A) - f(a_1) - f(a_2)$

# How to prove

$$\mathrm{OPT}_{\mathrm{ABL}}(A) = \mathrm{OPT}_{\mathrm{ABL}}(A') + f(a_1) + f(a_2)\ ?$$

We proved

- $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) \le \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}') + \boldsymbol{f}(\boldsymbol{a}_1) + \boldsymbol{f}(\boldsymbol{a}_2)$ (1)

- $\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}') \le \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) - \boldsymbol{f}(\boldsymbol{a}_1) - \boldsymbol{f}(\boldsymbol{a}_2)$

$$\Downarrow$$

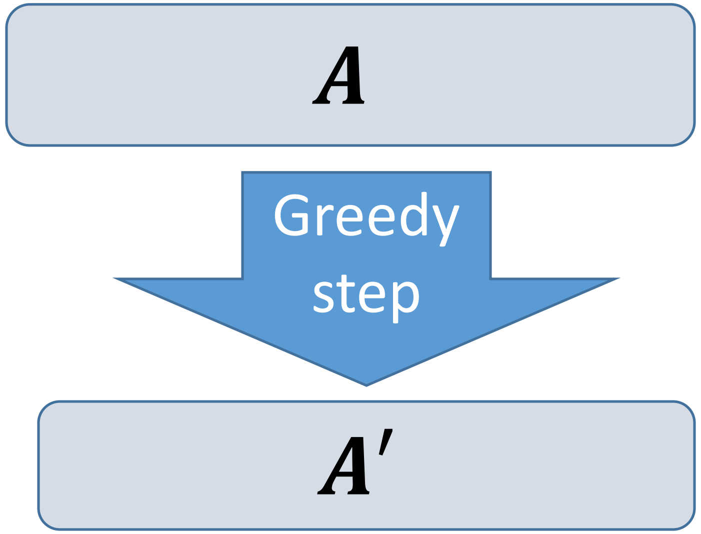
$$\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) \ge \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}') + \boldsymbol{f}(\boldsymbol{a}_1) + \boldsymbol{f}(\boldsymbol{a}_2) \qquad (2)$$

Using (1) and (2)

$$\boxed{\mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}) = \mathbf{OPT}_{\mathbf{ABL}}(\boldsymbol{A}') + \boldsymbol{f}(\boldsymbol{a}_1) + \boldsymbol{f}(\boldsymbol{a}_2)}$$

# To prove that a greedy strategy works

**P**: A given optimization problem




instance of size $n$ of problem **P**

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.

# Fractional Knapsack

**Input:**

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

**Output:**

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

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

# Question 5

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

Use the above as greedy strategy to design an algorithm for the fractional knapsack problem.

# 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