Greedy Algorithms - 1

CSIE 2136 Algorithm Design and Analysis, Fall 2018

https://cool.ntu.edu.tw/courses/61

Hsu-Chun Hsiao

Announcement

Homework assignments

- Mini-hw5 due next week
- HW2 due in 3 weeks

TA hour schedule

要期中考啦！！！(11/8)

期中範圍

- Divide and conquer
- Dynamic programming
- Greedy algorithms
- 當然也有recurrence and asymptotic notations

以上課投影片和作業為主

時間: 14:20-17:20, 3 hours

地點：102 & 103 & 104 (會排座位 請稍微提早抵達)

形式：是非/選擇、簡答、證明/說明

- Easy: ~65%, medium: ~25%, hard: ~10%

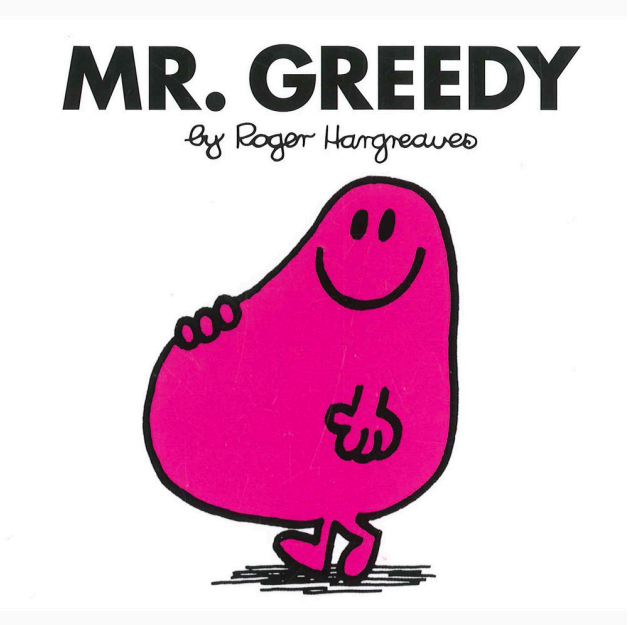
Closed book

Agenda

Greedy algorithms: definition and properties

Examples

- Coin changing
- Huffman encoding
- Interval scheduling
- Scheduling to minimize lateness



Intro to Greedy Algorithms

Greedy algorithms

Try to solve optimization problems by “greedy” choices

Always make a choice that looks best at the moment

Make a **locally optimal choice** in hope of getting a **globally optimal solution**

生活中greedy decisions的例子？

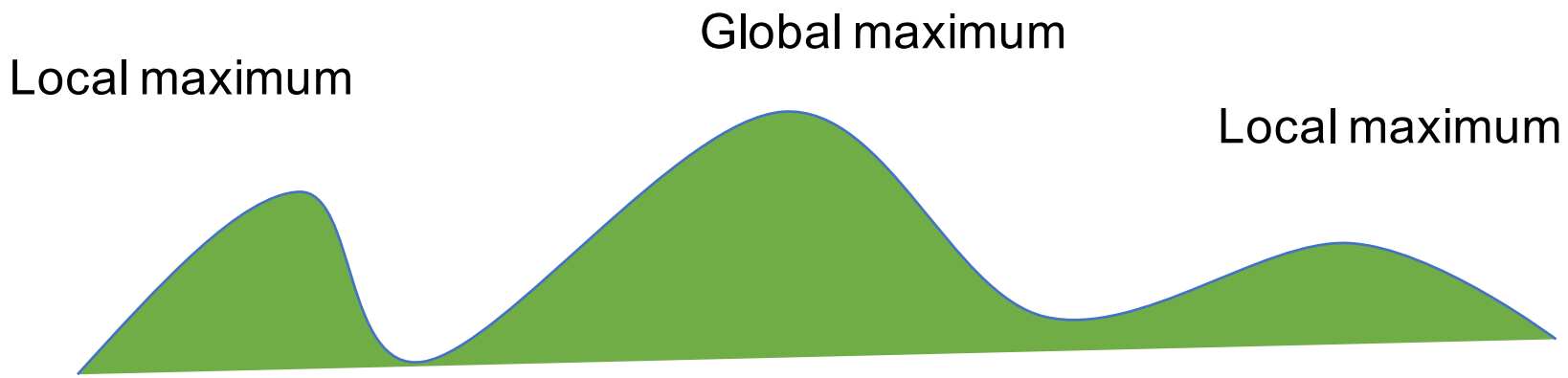
總是能獲得最佳解嗎？

Greedy algorithms

Many “greedy” algorithms to same problem

- Easy to invent one
- Do not always yield optimal solutions
- Hard to find one that actually works and prove its optimality

Example of greedy choice:
move toward max gradient => may end up at a local optimal



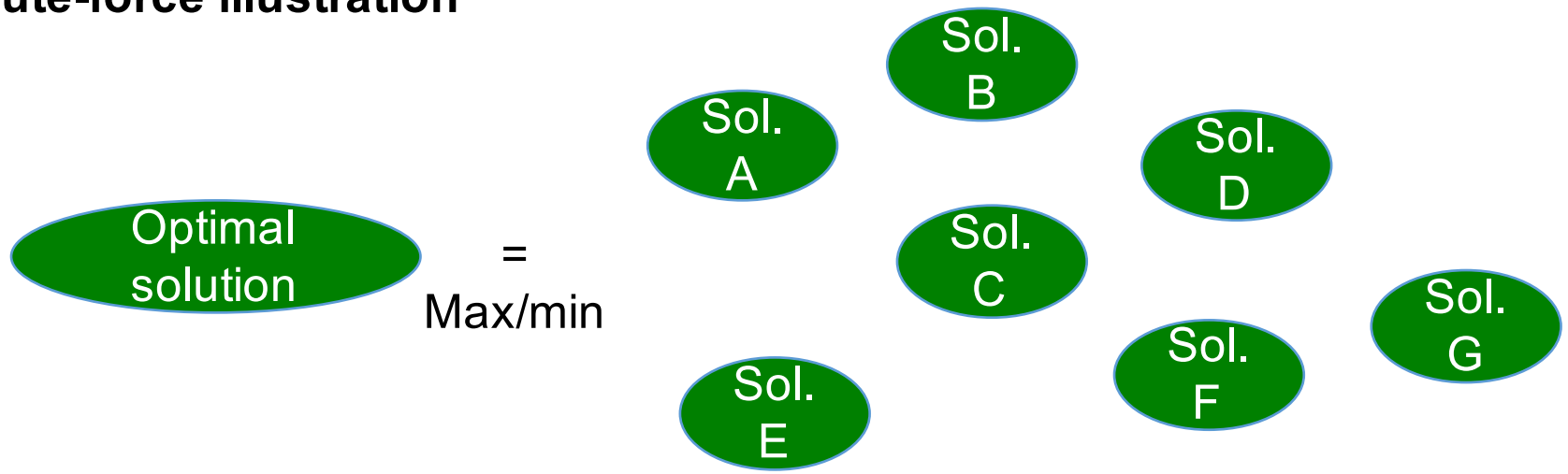
Greedy algorithms

To yield an optimal solution, the problem should exhibit

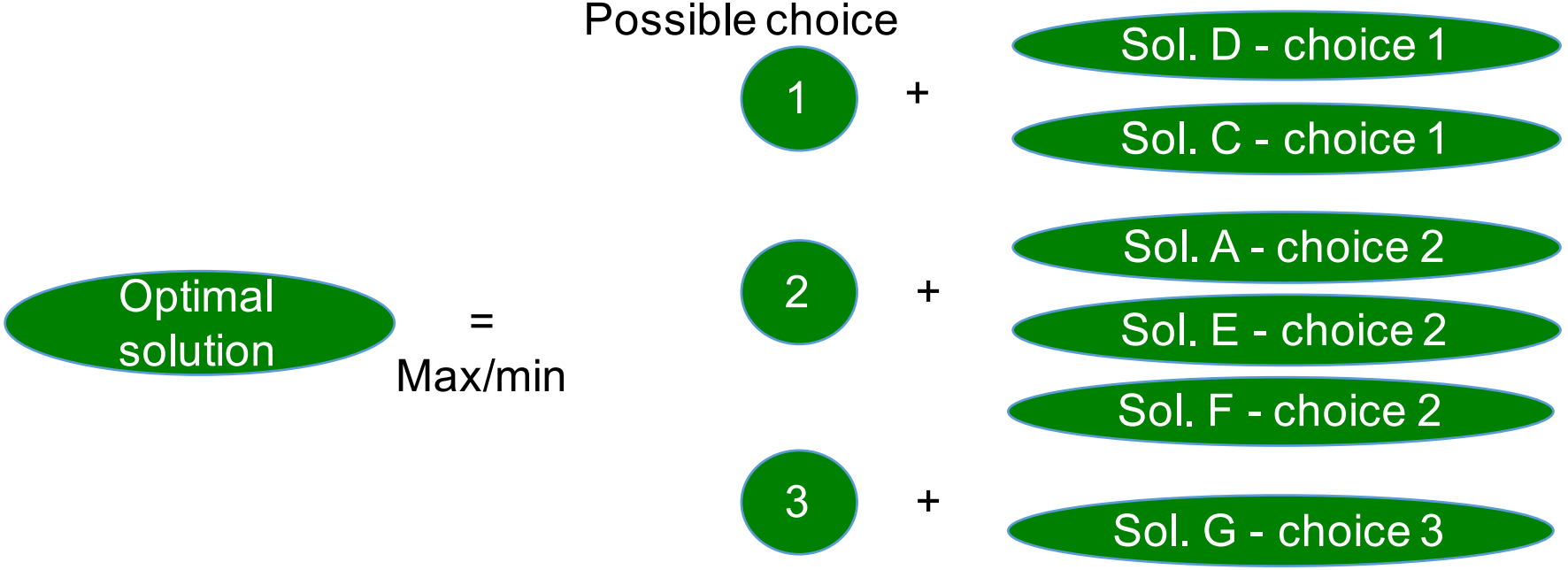
1. **Greedy-choice property**
   - Making locally optimal (greedy) choices leads to a globally optimal solution
2. **Optimal substructure**
   - An optimal solution to the problem contains within it optimal solutions to subproblems

Later we will learn how to prove these two properties

Brute-force illustration

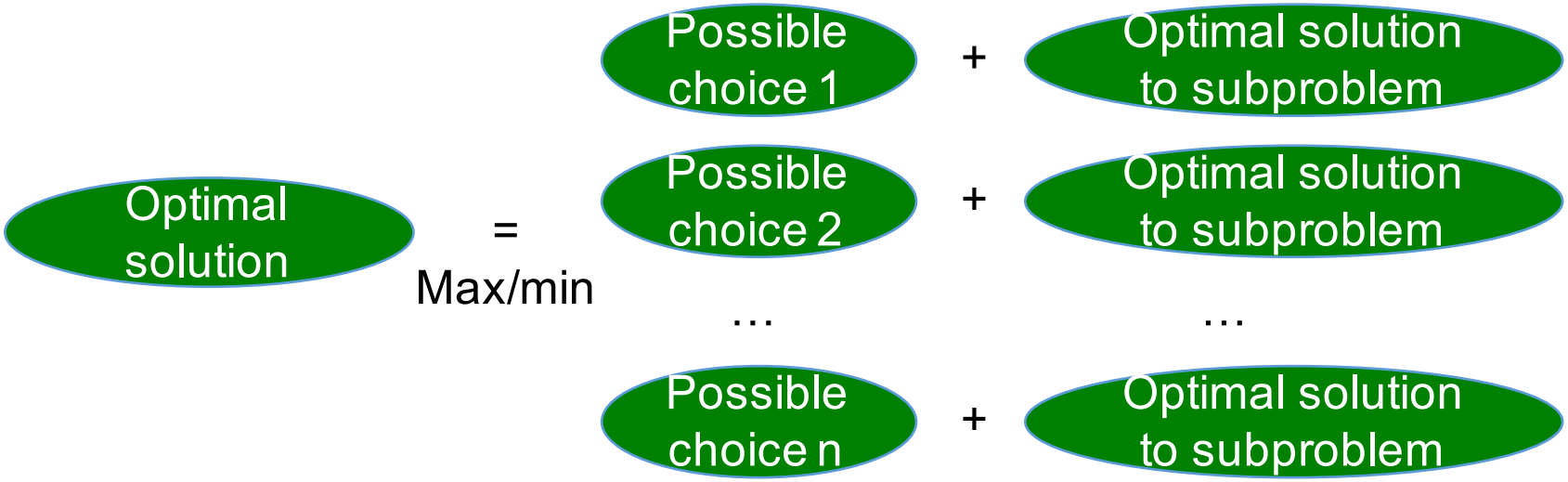


Same brute-force illustration (grouping solutions by exclusive choices)



DP illustration

Optimal substructure property ensures that we only need to consider an optimal solution to each subproblem



對每個可能的情況，只要考慮相對應的subproblem的一個optimal解就可以了。
其他的solution都不用考慮了！

Greedy illustration

Greedy-choice property ensures that we only need to consider one greedy choice (among all possible choices)

Dynamic Programming vs. Greedy

Dynamic programming **Greedy algorithms**

Both require **optimal substructure**

- Make an informed choice after getting optimal solutions to subproblems
- Overlapping subproblems

- Make a greedy choice before solving the resulting subproblem
- No overlapping subproblem
  - Each round selects only one subproblem
  - Sizes of subproblems decrease

Coin Changing (找零錢)

Textbook Problem 16-1

百萬小學堂－資訊（二年級）

目前常用的貨幣面額為$1, $5, $10, $50

假設硬幣數量充足，要如何使用最少的硬幣找錢給顧客？

Ex. $104

6枚

Ex. $27

5枚

Cashier's algorithm (收銀員演算法): At each iteration, add coin of the largest possible value without exceeding the total

Cashier's algorithm

Greedy choice: at each iteration, add coin of the largest possible value without exceeding the total

Is cashier's algorithm optimal?

Proof

Cashier's algorithm

Optimal substructure

- Let coins[n] be the minimum number of coins to pay n
- Suppose we know OPT to coins[x]
- Case 1: OPT contains Coin 1
  - coins[x] = 1 + coins[x-c_1]
- Case 2: OPT contains Coin 2
  - coins[x] = 1 + coins[x-c_2]
- Case 3: OPT contains Coin 3
  - coins[x] = 1 + coins[x-c_3]
- Case 4: OPT contains Coin 4
  - coins[x] = 1 + coins[x-c_4]

Cashier's algorithm

Optimal substructure

- Let coins[n] be the minimum number of coins to pay n

DP way

Coins[x] = min


+ Coins[x-50]


+ Coins[x-10]


+ Coins[x-5]


+ Coins[x-1]

Cashier's algorithm

Greedy choice: at each iteration, add coin of the largest possible value without exceeding the total

- 四種可能，但greedy choice直接選定一種
- 如何說明存在OPT solution包含此greedy choice？

Greedy way

When 50 > x >=10

Coins[x] = min


+ Coins[x-50]


+ Coins[x-10]


+ Coins[x-5]


+ Coins[x-1]

Cashier's algorithm

Greedy choice: at each iteration, add coin of the largest possible value without exceeding the total

- 四種可能，但greedy choice直接選定一種
- 如何說明存在OPT solution包含此greedy choice？

Greedy-choice property

- 反證法：假設不存在OPT解包含此greedy choice，推出矛盾結論
- 考慮當 50 > x >= 10的狀況 （其他狀況也可類推）
- 根據反證法假設，所有的OPT解都只能用50元, 5元, 1元湊成
- 50元用不到
- 5元不能超過1個（不然就換10元就好了）
- 1元不能超過4個（不然就換5元就好了）
- => 湊不到x元，矛盾！

Proof of correctness

Several proof techniques; here are some examples:

- **Greedy algorithm stays ahead**: Prove by induction that the algorithm outputs a better solution than others at each inductive step
- **Exchange argument**: Transform solution into another optimal solution without hurting its quality

In this class, we will focus on proving the following two properties:

1. **Greedy-choice property**
   - Making locally optimal (greedy) choices leads to a globally optimal solution
2. **Optimal substructure**
   - An optimal solution to the problem contains within it optimal solutions to subproblems

Proof of correctness: Greedy-choice property

Making locally optimal (greedy) choices leads to a globally optimal solution

- Show that $\exists$ an optimal solution which "contains" the greedy choice
- This shows that we are on the right track to get to an optimal solution

Usually established by ***exchange argument***

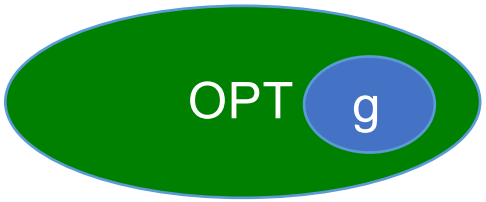
Proof of correctness: Greedy-choice property

Exchange argument

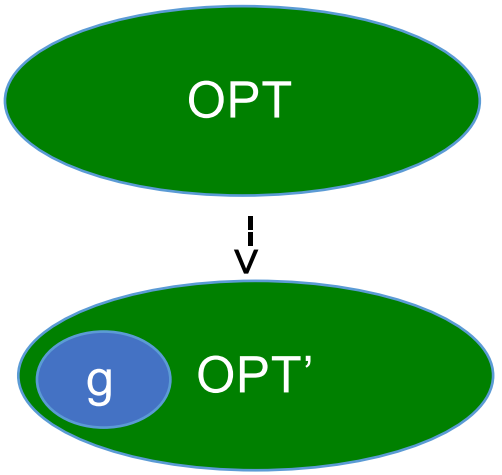
For ***any*** optimal solution OPT, we know that the greedy choice g is either in OPT or not in OPT

- If OPT contains g, then done
- If OPT does not contain g, show that we can modify OPT into OPT' such that OPT' contains g and is ***at least as good as*** OPT

g in OPT



g not in OPT



Proof of correctness: Greedy-choice property

Making locally optimal (greedy) choices leads to a globally optimal solution

- Show that $\exists$ an optimal solution which "contains" the greedy choice

Usually established by ***exchange argument***

1. For any optimal solution OPT, we know that the greedy choice g is either in OPT or not in OPT
2. If OPT contains g, then done
3. If OPT does not contain g, show that we can modify OPT into OPT' such that OPT' contains g and is at least as good as OPT

- OPT' is usually obtained by **exchanging** some element in OPT with g
- If OPT' is better than OPT (every optimal solution contains g), the property is proved by contradiction
- If OPT' is as good as OPT (some optimal solutions do not contain g), then we showed that there exists an optimal solution containing p by construction

Other set of denominations?

Cashier's algorithm may not work for other denominations

Find a counterexample to prove that cashier's algorithm doesn't work in the following settings:

How about $1, $6, $10?

How about $5, $6, $7?

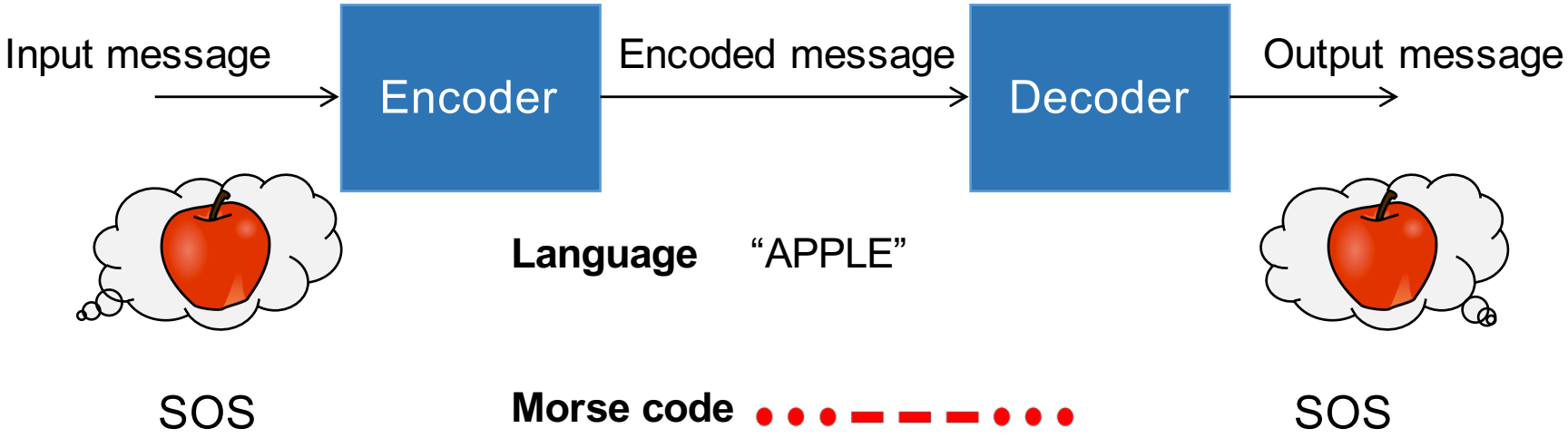
Huffman Codes (霍夫曼編碼)

Textbook Chapter 16.3

Chapter 4.8 in Algorithm Design by Kleinberg & Tardos

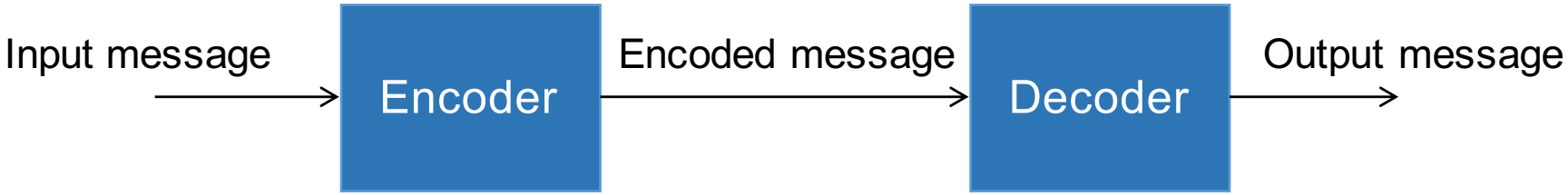
Background: message encoding/decoding

"Code (編碼) is system of rules to convert information—such as a letter, word, sound, image, or gesture—into another, sometimes shortened or secret, form or representation for communication through a channel or storage in a medium."



Background: message encoding/decoding

"Code (編碼) is system of rules to convert information—such as a letter, word, sound, image, or gesture—into another, sometimes shortened or secret, form or representation for communication through a channel or storage in a medium."



Why encoding?

- Enable communication or storage
- Detect or correct errors introduced during transmission
- Compress data (lossily or losslessly)
  - E.g., Huffman coding

Background: Lossless data compression

Problem: encode symbols using bits

- How to represent these symbols?
- How to minimize the number of bits?
- How to ensure decode(encode(x)) = x?

Fixed-length code vs. variable-length code

- Fixed-length: encodes every symbol using an equal number of bits
- Variable-length: assigns shorter codewords to more frequent symbols

Symbols	a	b	c	d	e	f
Frequency	0.45	0.13	0.12	0.16	0.09	0.05
Code 1 (fixed-length codeword)	000	001	010	011	100	101
Code 2 (variable-length codeword)	0	101	100	111	1101	1100

In Code 1, encode(ad) = 000 011

In Code 2, encode(ad) = 0 111

Background: Lossless data compression

Problem: encode symbols using bits

- How to represent these symbols?
- How to minimize the number of bits?
- How to ensure decode(encode(x)) = x?

Fixed-length code vs. variable-length code

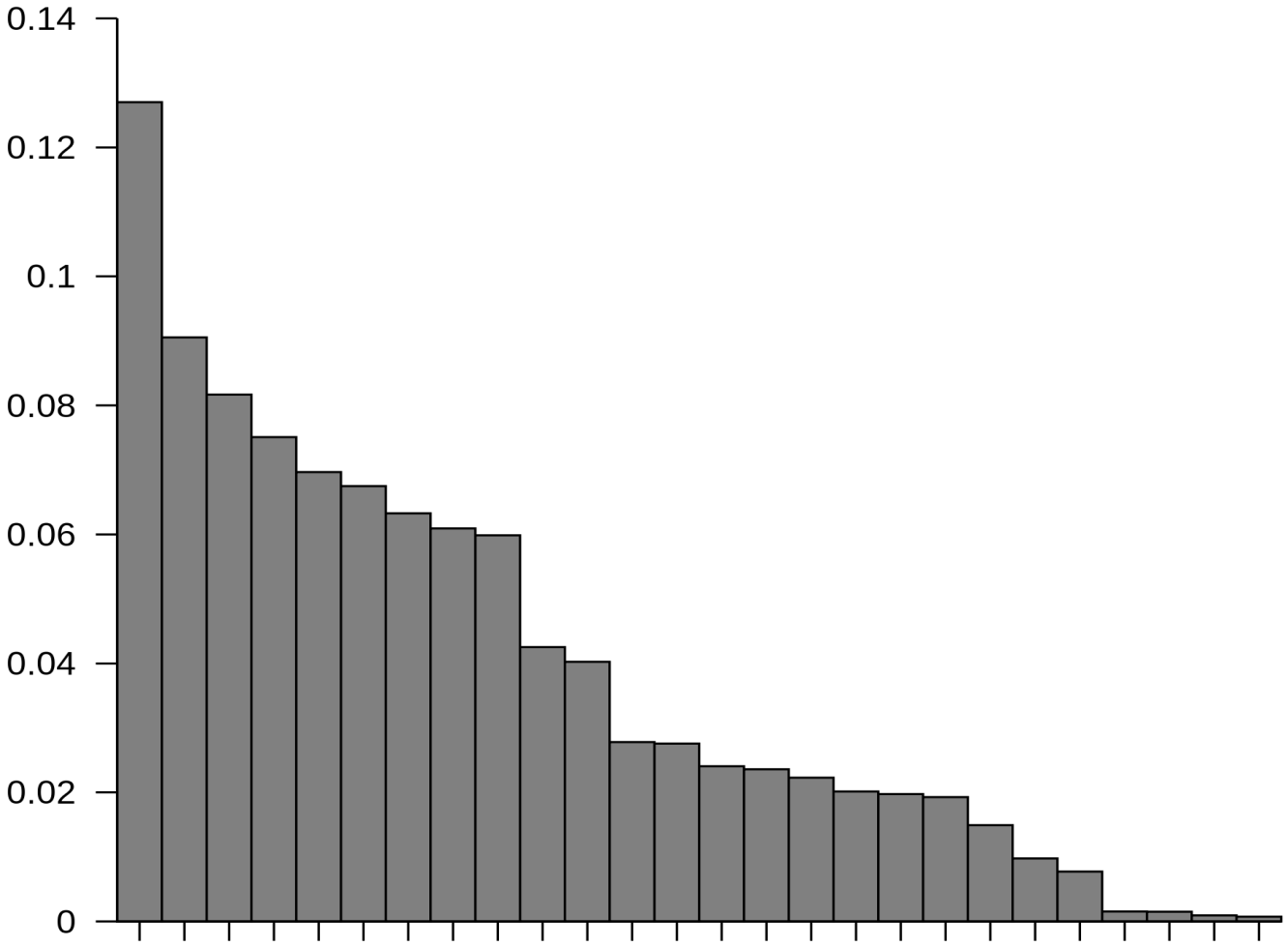
- Fixed-length: encodes every symbol using an equal number of bits
- Variable-length: assigns shorter codewords to more frequent symbols

Symbols	a	b	c	d	e	f
Frequency	0.45	0.13	0.12	0.16	0.09	0.05
Code 1 (fixed-length codeword)	000	001	010	011	100	101
Code 2 (variable-length codeword)	0	101	100	111	1101	1100

Average # of bits per symbol in Code 1 = 3

Average # of bits per symbol in Code 2 = 0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4 = 2.24

Letter frequency in English alphabet (字母頻率)



Background: prefix codes

A special type of variable-length codes

No codeword is a prefix of some other codeword

- Prefix codes are uniquely decodable

Symbols	a	b	c	d	e	f
Frequency	0.45	0.13	0.12	0.16	0.09	0.05
Code 2 (variable-length & prefix code)	0	101	100	111	1101	1100
Code 3 (variable-length but not prefix code; c is prefix of b)	0	101	10	111	1101	1100

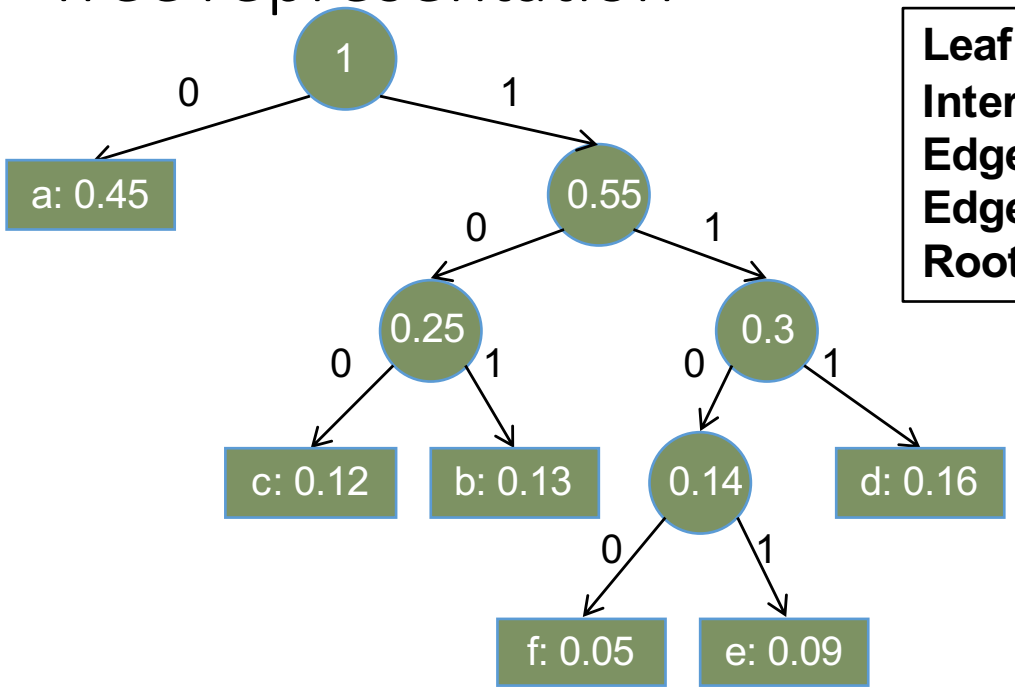
In Code 3, decode(1011100) = ‘bf’ or ‘cdaa’

Background: prefix codes

No codeword is a prefix of some other codeword

Symbols	a	b	c	d	e	f
Frequency	0.45	0.13	0.12	0.16	0.09	0.05
Code 2 (variable-length & prefix code)	0	101	100	111	1101	1100

Tree representation



Leaf node: symbol and frequency
Internal node: sum of children's frequencies
Edge to left child: 0
Edge to right child: 1
Root to leaf path: codeword

Decoding is simple using this tree

decode(110001001101) = ?

Optimal prefix codes

Optimal prefix code = lowest average bits per symbol

Define cost of a prefix tree, B(T)

= average depth per leaf node

= average bits per symbol

$$B(T) = \sum_{s \in S} freq(s) \cdot d_T(s)$$

S = set of symbols
freq(s) = frequency of s
d_T(s) = depth of s in tree T

We will show that Huffman codes are optimal prefix codes

- Prove that given a set of symbols and their frequencies, Huffman coding produces a prefix tree with a minimal B(T)

Huffman codes (霍夫曼編碼法)

Invented by Huffman in 1950

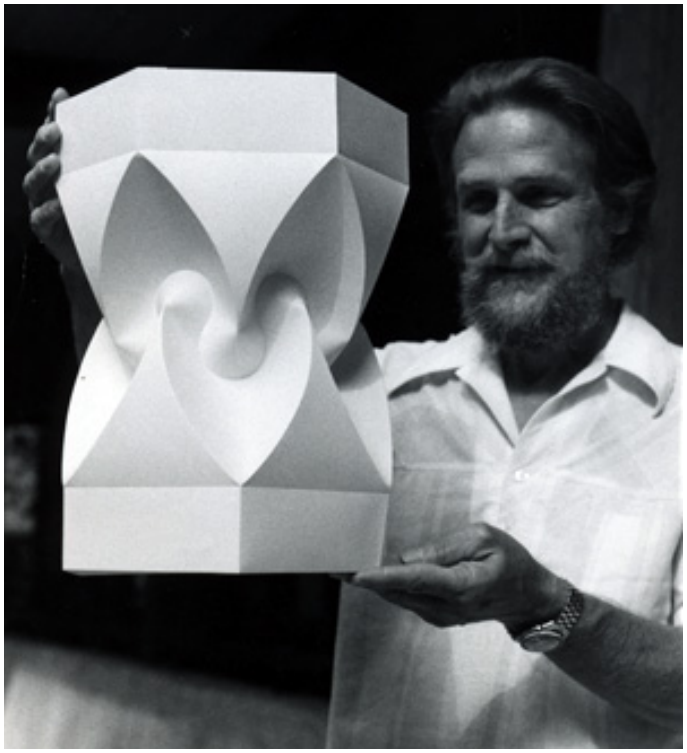
- Class assignment of MIT information theory class

Lossless data compression

Optimal prefix code

Efficient to generate codewords

Efficient to encode and decode



David A. Huffman
Photo from UC Santa Cruz press release

Codeword generation

Start with a forest of n trees each consisting of a single node corresponding to a symbol s and with frequency freq(s)

Greedy choice: merge repeatedly until one tree left:

- Select two trees x, y with minimal frequency roots freq(x) and freq(y)
- Join into single tree by adding root z with frequency freq(x) + freq(y)

Example:

Initial set (stored in a priority queue)

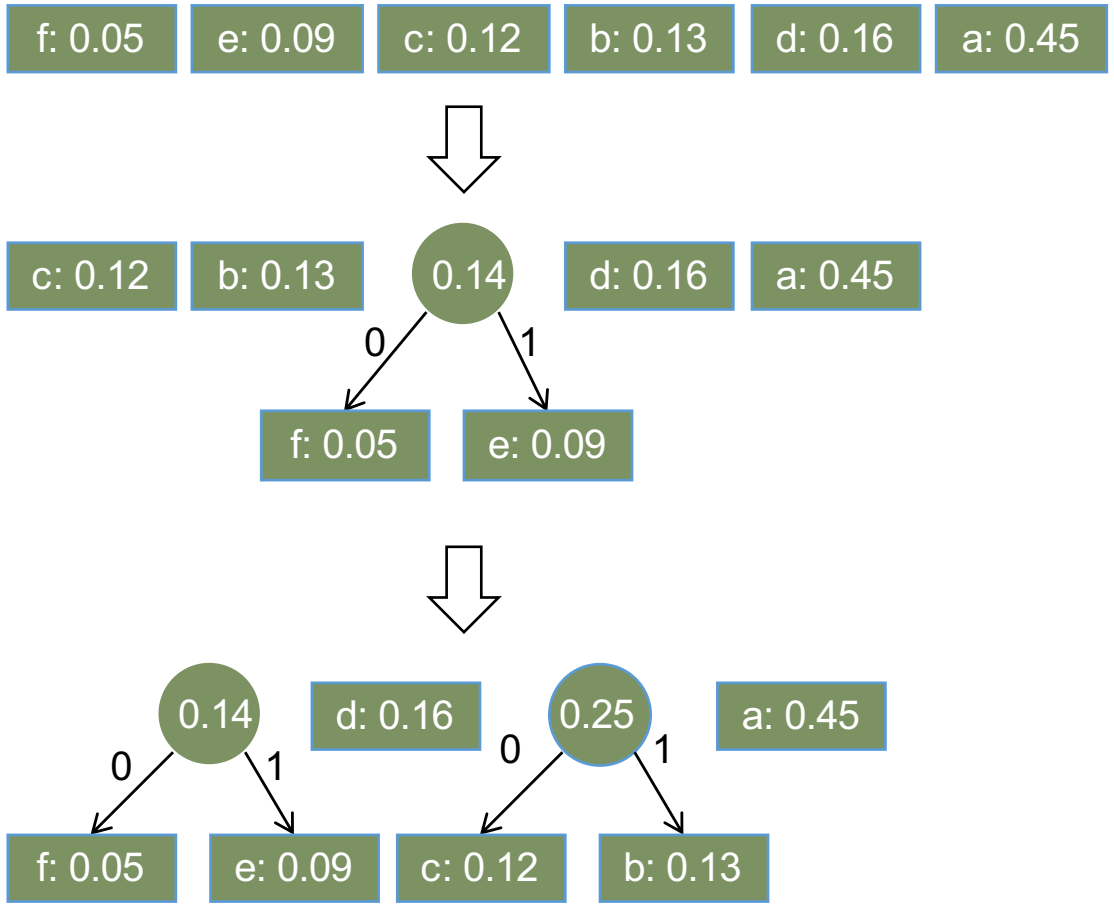
f: 0.05	e: 0.09	c: 0.12	b: 0.13	d: 0.16	a: 0.45

Greedy choice: merge repeatedly until one tree left:

- Select two trees x, y with minimal frequency roots freq(x) and freq(y)
- Join into single tree by adding root z with frequency freq(x) + freq(y)

Example:

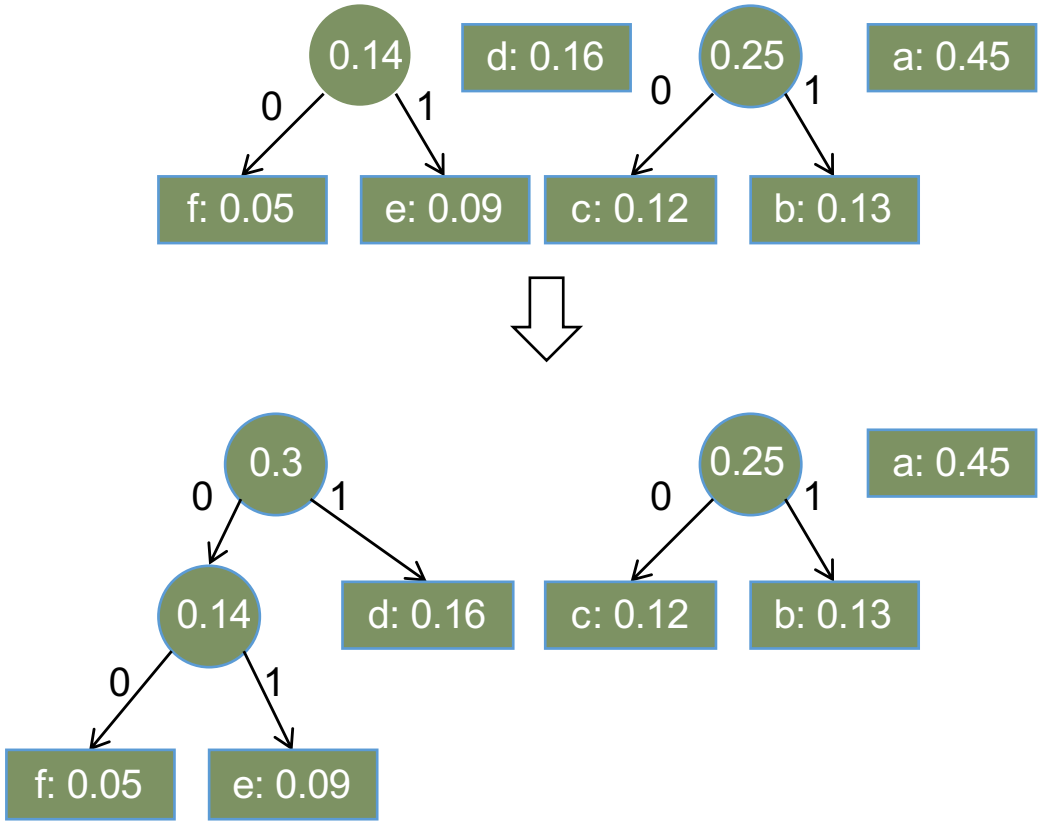
Initial set (stored in a priority queue)



Greedy choice: merge repeatedly until one tree left:

- Select two trees x, y with minimal frequency roots freq(x) and freq(y)
- Join into single tree by adding root z with frequency freq(x) + freq(y)

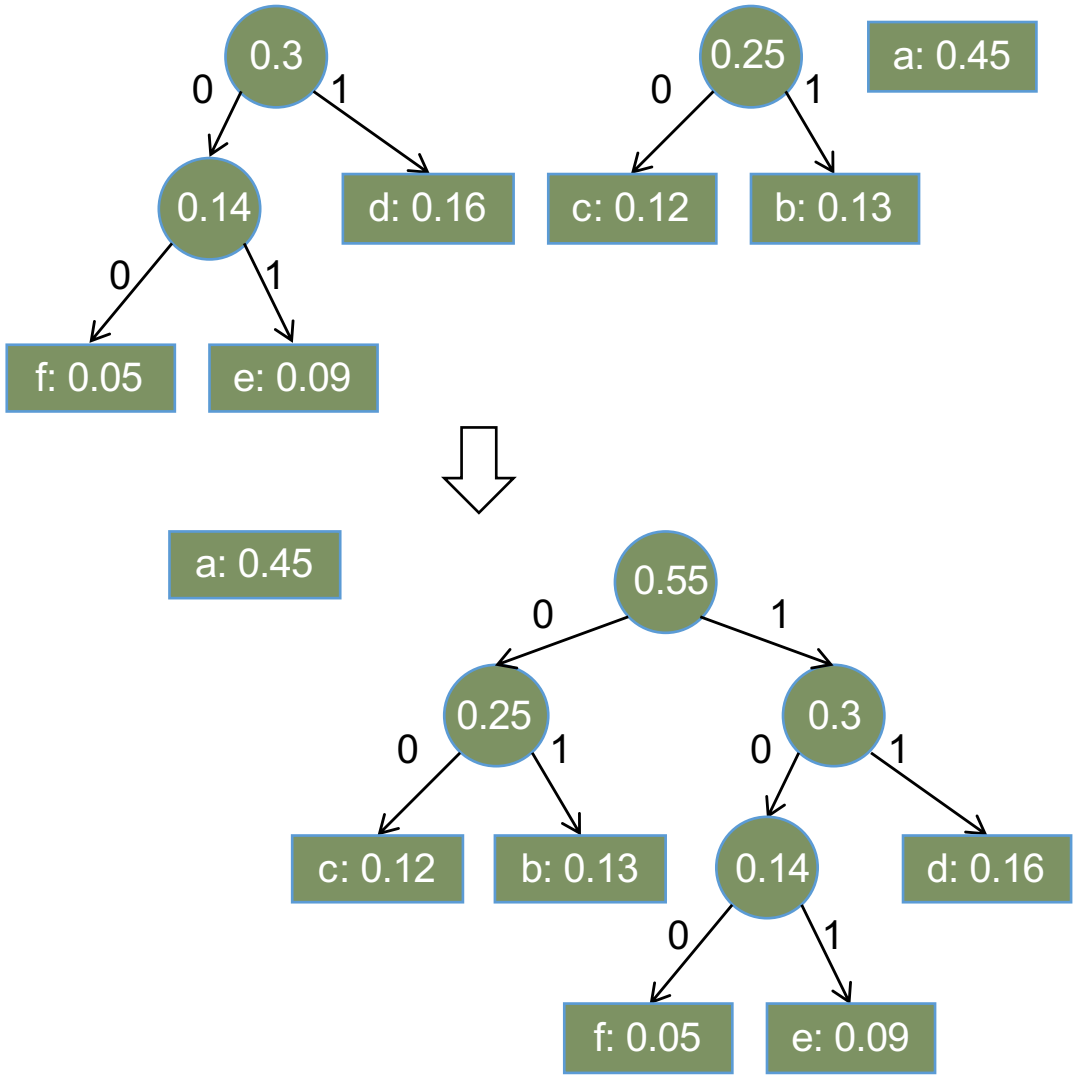
Example:



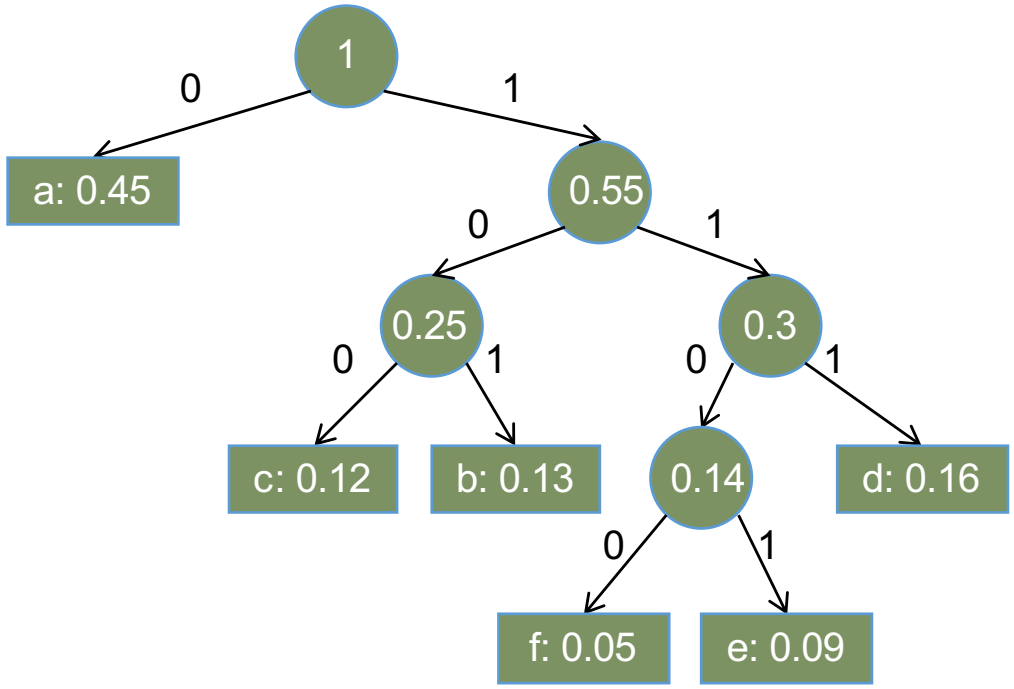
Greedy choice: merge repeatedly until one tree left:

- Select two trees x, y with minimal frequency roots freq(x) and freq(y)
- Join into single tree by adding root z with frequency freq(x) + freq(y)

Example:



Running time analysis




```
Huffman(S):
    n = |S|
    Q = Build-Priority-Queue(S)
    For i = 1 to n-1
        allocate a new node z
        z.left = x = Extract-Min(Q)
        z.right = y = Extract-Min(Q)
        freq(z) = freq(x) + freq(y)
        Insert(Q, z)
    Return Extract-Min(Q) //return prefix tree
```

Running time = O(nlogn)

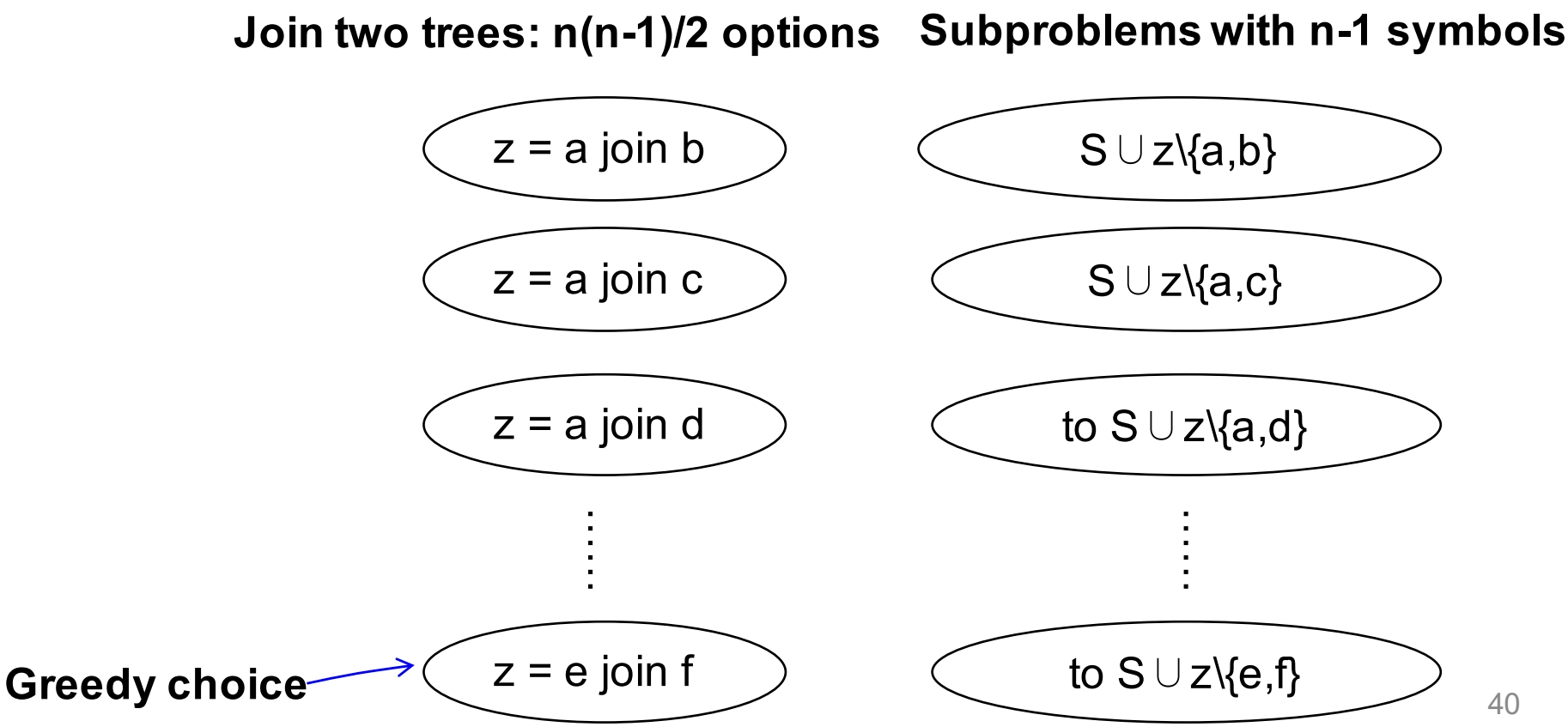
Suppose the priority queue supports:
Build-Priority-Queue(S) = O(nlogn)
Extract-Min(Q) = O(logn)
Insert(Q, z) = O(logn)

Greedy choice and subproblems

What are the options among which we make a greedy choice?

What is the subproblem after the greedy choice is made?

f: 0.05	e: 0.09	c: 0.12	b: 0.13	d: 0.16	a: 0.45



Proof of greedy choice property

Let x & y be two symbols in S having the lowest frequencies

Prove that $\exists$ an optimal prefix code for S in which the codewords for x & y have the same length and differ only in the last bit

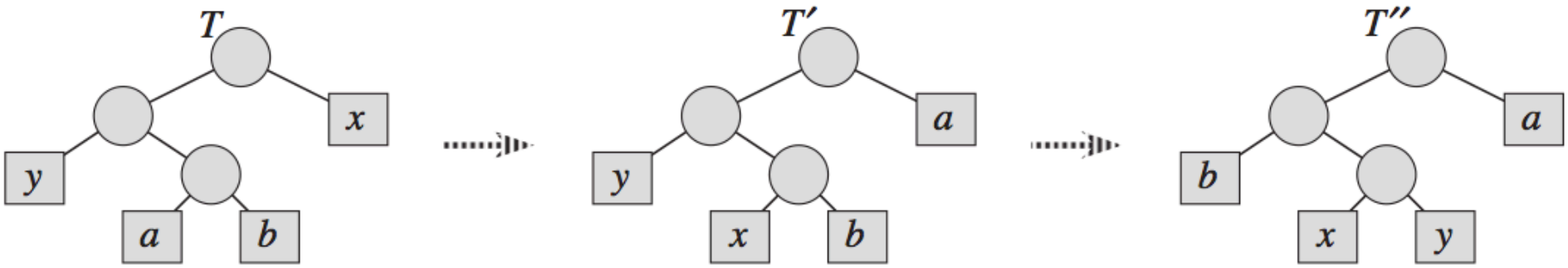
Proof by the exchange argument:

Key idea: suppose tree T representing an arbitrary optimal code. Modify T into another optimal code such that symbols x and y appear as sibling leaves of max-depth in the new tree.

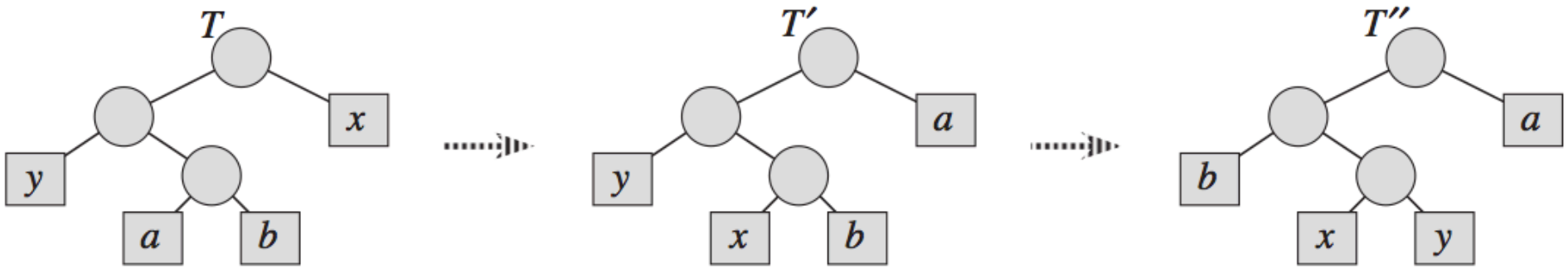
Let a, b be the two sibling leaves of maximum depth in T

WLOG, assume freq(a) ≤ freq(b) and freq(x) ≤ freq(y)

Because freq(x) and freq(y) are the two lowest leaf frequencies, we know freq(x) ≤ freq(a) and freq(y) ≤ freq(b)



Proof of greedy choice property



$$B(T) - B(T')$$

$$= \sum_{s \in S} freq(s) \cdot d_T(s) - \sum_{s \in S} freq(s) \cdot d_{T'}(s)$$

S = set of symbols
freq(s) = frequency of s
$d_T(s)$ = depth of s in tree T

$$= freq(x) \cdot d_T(x) + freq(a) \cdot d_T(a) - freq(x) \cdot d_{T'}(x) - freq(a) \cdot d_{T'}(a)$$

$$= freq(x) \cdot d_T(x) + freq(a) \cdot d_T(a) - freq(x) \cdot d_T(a) - freq(a) \cdot d_T(x)$$

$$= (freq(a) - freq(x))(d_T(a) - d_T(x))$$

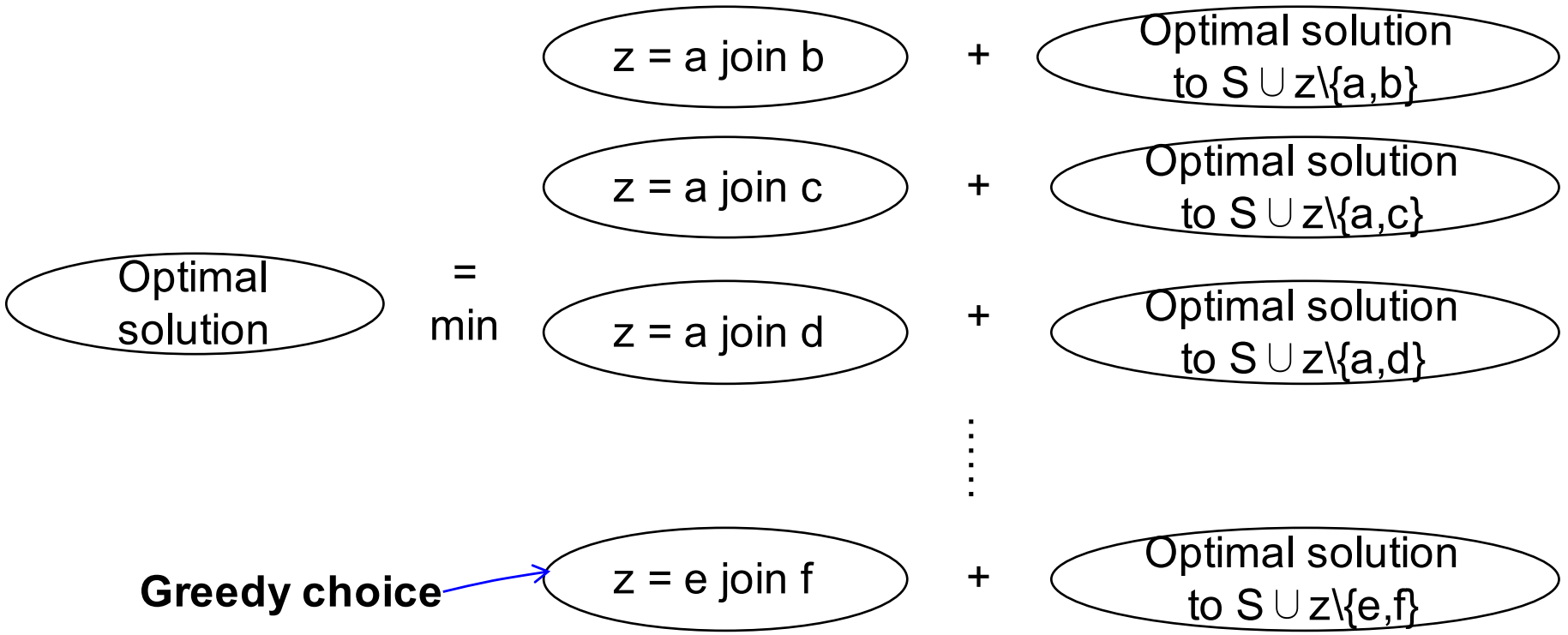
$$\geq 0$$

Practice 1: prove that B(T') – B(T'') ≥ 0
Practice 2: prove that an optimal prefix tree must be a full tree

Proof of optimal substructure

Let's prove the optimal substructure for all cases

Alternatively, we can also take advantage of greedy choice and prove optimal substructure only for the greedy case



Proof of optimal substructure

Let T be a full binary tree representing an optimal prefix code for S

Let x and y be any leaves of T which share the same parent z

Let S' = S\{x, y}$\cup${z}, with freq(z) = freq(x) + freq(y)

Prove that T' = T\{x, y} is an optimal tree for S'

Proof by contradiction:

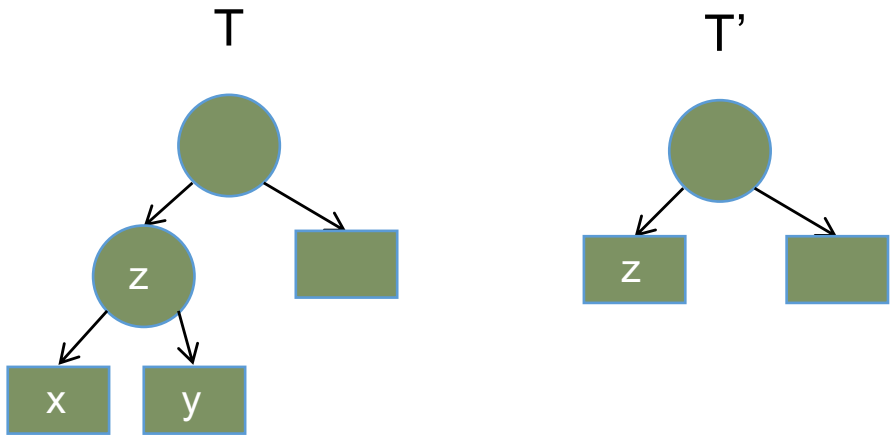
Suppose T' is not optimal for S'

$\exists$ T'' is better than T' for S'

Then T''$\cup${x, y} is better than T for S (why?)

Contradiction!

$$
\begin{aligned}
& B(T'') + freq(x) + freq(y) \\
& < B(T') + freq(x) + freq(y) \\
& = B(T)
\end{aligned}
$$



Proof of optimal substructure

Alternatively, consider one branch (greedy choice) only

- Sometimes this is simpler and easier to think

Prove the greedy case has optimal substructure

- Let T be a full binary tree representing an optimal prefix code for S
- Let x and y be two symbols in T with minimum frequencies
- Let S' = S\{x, y} $\cup$ {z}, with freq(z) = freq(x) + freq(y)
- Prove that T' = T\{x, y} is an optimal tree for S'

Practice by yourself

Optimality

Theorem: Huffman algorithm generates an optimal prefix code

Proof by induction on the number of symbols n

Induction hypothesis: Huffman codes are optimal for n symbols

Base case: n = 2, optimal

Consider a set S with n+1 symbols

1. By construction, we know that the two symbols with minimum frequencies are siblings in T
2. Construct T' by replacing the two symbols x & y with symbol z such that $S' = (S \setminus \{x, y\}) \cup \{z\}$ and $freq(z) = freq(x) + freq(y)$
3. T' is optimal by induction
4. Because of the optimal substructure property & greedy choice property, we know that when T' is optimal, T is optimal too

Once we have proved the greedy choice property and the optimal substructure property, this induction proof framework can be applied to prove its optimality. We will omit this step from now on.

Drawbacks of Huffman codes

Huffman's algorithm is optimal for a symbol-by-symbol coding with a known input probability distribution

Huffman's algorithm may be sub-optimal when

- Blending among symbols is allowed
- The probability distribution is unknown
- Symbols are not independent

freq(a) = 0.5 and freq(b) = 0.5

- Huffman code? cost?

freq(a) = 0.999 and freq(b) = 0.001

- Huffman code? cost?
- Any better representation?