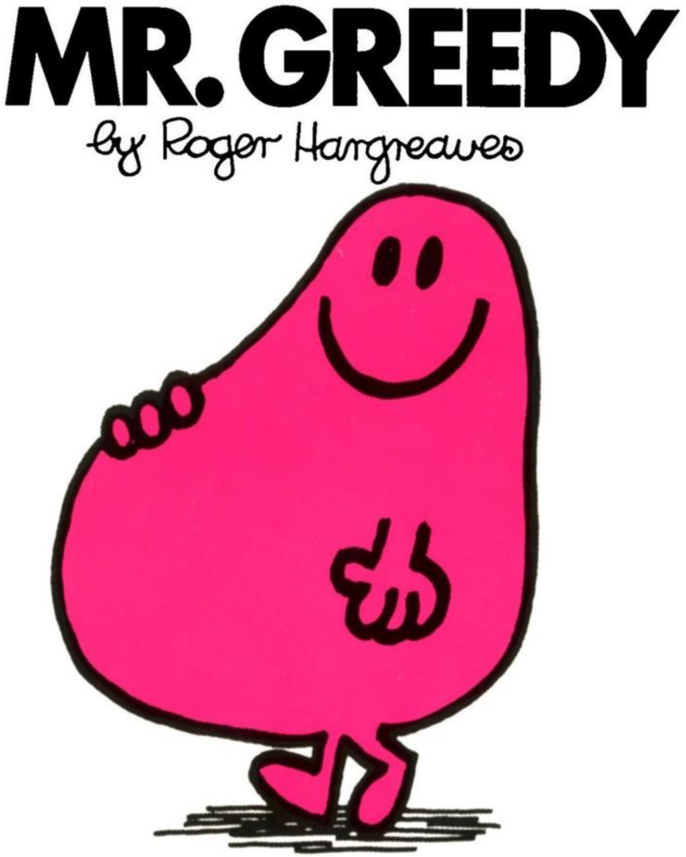


Algorithm Design and Analysis
Greedy Algorithms

http://ada.miulab.tw slido: #ADA2021

Yun-Nung (Vivian) Chen

國立臺灣大學
National Taiwan University

Outline

- Greedy Algorithms
- Greedy #1: Activity-Selection / Interval Scheduling
- Greedy #2: Coin Changing
- Greedy #3: Fractional Knapsack Problem
- Greedy #4: Huffman Codes
- Greedy #5: Breakpoint Selection
- Greedy #6: Task-Scheduling
- Greedy #7: Scheduling to Minimize Lateness

Algorithm Design Strategy

- Do not focus on "specific algorithms"
- But "some strategies" to "design" algorithms

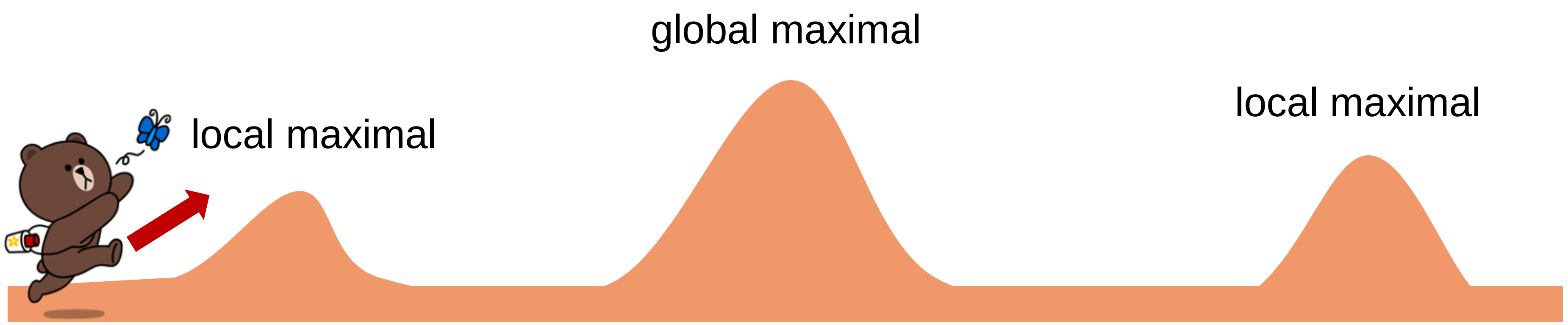
- First Skill: Divide-and-Conquer (各個擊破/分治)
- Second Skill: Dynamic Programming (動態規劃)
- Third Skill: Greedy (貪婪法則)

Greedy Algorithms

Textbook Chapter 16 – Greedy Algorithms
Textbook Chapter 16.2 – Elements of the greedy strategy

What is Greedy Algorithms?

- always makes the choice that looks best at the moment
- makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
  - not always yield optimal solution; may end up at local optimal



Greedy: move towards max gradient and hope it is global maximum

Algorithm Design Paradigms

- Dynamic Programming
  - has **optimal substructure**
  - make an informed choice after getting optimal solutions to subproblems
  - **dependent** or **overlapping** subproblems

Optimal Solution = max /min {
Possible Case 1 + Subproblem Solution
Possible Case 2 + Subproblem Solution
Possible Case k + Subproblem Solution
}

- Greedy Algorithms
  - has **optimal substructure**
  - make a greedy choice before solving the subproblem
  - **no overlapping** subproblems
    - ✓ Each round selects only one subproblem
    - ✓ The subproblem size decreases

Optimal Solution = Greedy Choice + Subproblem Solution

Greedy Procedure

1. Cast the optimization problem as one in which we make a choice and remain one subproblem to solve
2. Demonstrate the optimal substructure
   - ✓ Combining an optimal solution to the subproblem via greedy can arrive an optimal solution to the original problem
3. Prove that there is always an optimal solution to the original problem that makes the greedy choice

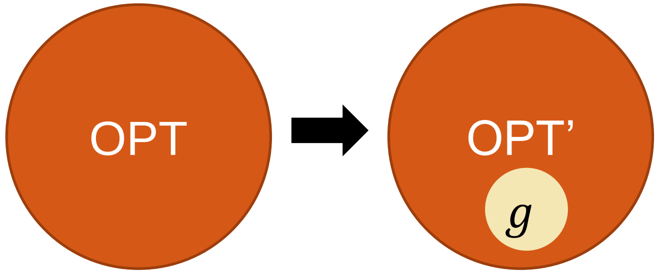
Greedy Algorithms

To yield an optimal solution, the problem should exhibit

1. Optimal Substructure : an optimal solution to the problem contains within its optimal solutions to subproblems
2. Greedy-Choice Property : making locally optimal (greedy) choices leads to a globally optimal solution

Proof of Correctness Skills

- Optimal Substructure: an optimal solution to the problem contains within its optimal solutions to subproblems
- Greedy-Choice Property: making locally optimal (greedy) choices leads to a globally optimal solution
  - Show that it exists an optimal solution that "contains" the greedy choice using **exchange argument**
  - For any optimal solution OPT, the greedy choice g has two cases
    - g is in OPT: done
    - g not in OPT: modify OPT into OPT' s.t. OPT' contains g and is at least as good as OPT



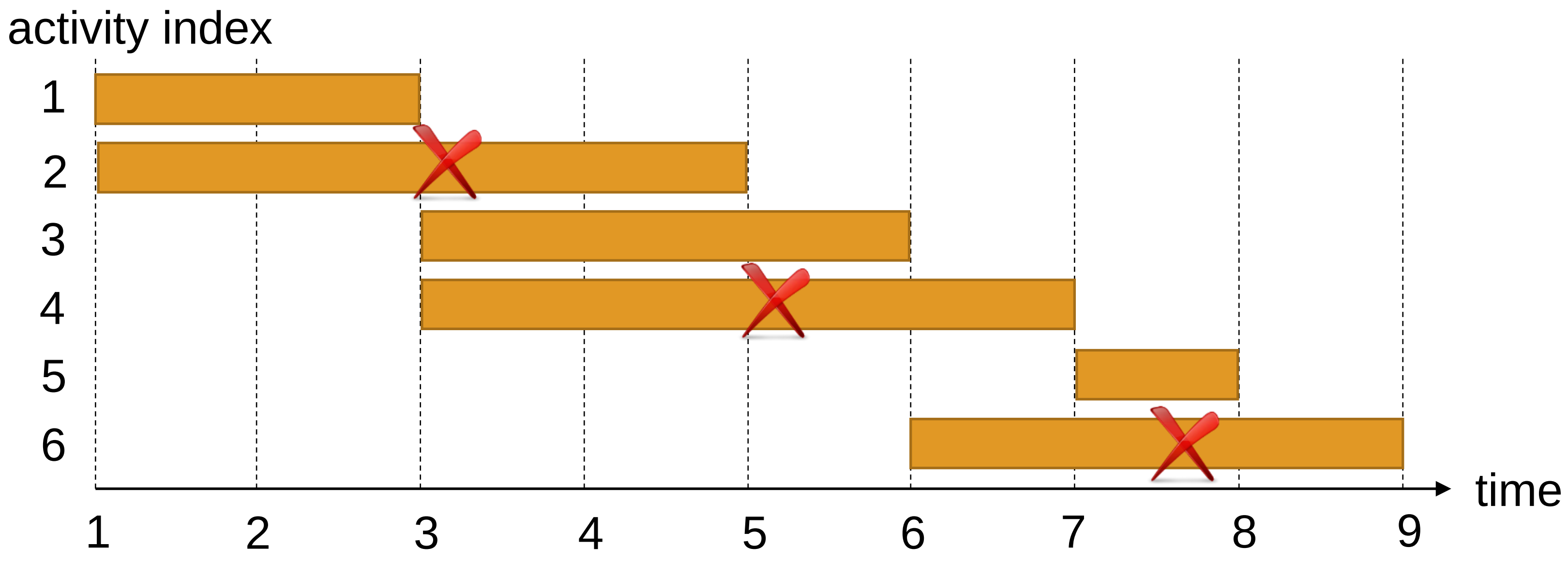
- ✓ If OPT' is better than OPT, the property is proved by contradiction
- ✓ If OPT' is as good as OPT, then we showed that there exists an optimal solution containing g by construction

Greedy #1: Activity-Selection / Interval Scheduling

Textbook Chapter 16.1 – An activity-selection problem

Activity-Selection/ Interval Scheduling

- Input: n activities with start times s_i and finish times f_i (the activities are sorted in monotonically increasing order of finish time $f_1 \leq f_2 \leq \cdots \leq f_n$)
- Output: the maximum number of compatible activities
- Without loss of generality: $s_1 < s_2 < \cdots < s_n$ and $f_1 < f_2 < \cdots < f_n$
  - 大的包小的則不考慮大的 → 用小的取代大的一定不會變差



Weighted Interval Scheduling

Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the <u>maximum total value</u> obtainable from compatible

- Subproblems
  - `WIS(i)`: weighted interval scheduling for the first i jobs
  - Goal: `WIS(n)`
- Dynamic programming algorithm

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + M_{p(i)}, M_{i-1}) & \text{otherwise} \end{cases}$$

i	0	1	2	3	4	5	…	n
M[i]								

$$T(n) = \Theta(n)$$

Set $v_i = 1$ for all i to formulate it into the activity-selection problem

Activity-Selection Problem

Activity-Selection Problem
Input: n activities with $\langle s_i, f_i \rangle$, $p(j)$ = largest index $i < j$ s.t. i and j are compatible
Output: the maximum number of activities

- Dynamic programming

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ \max(1 + M_{p(i)}, M_{i-1}) & \text{otherwise} \end{cases}$$

  - **Optimal substructure** is already proved

- Greedy algorithm

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 + M_{p(i)} & \text{otherwise} \end{cases}$$

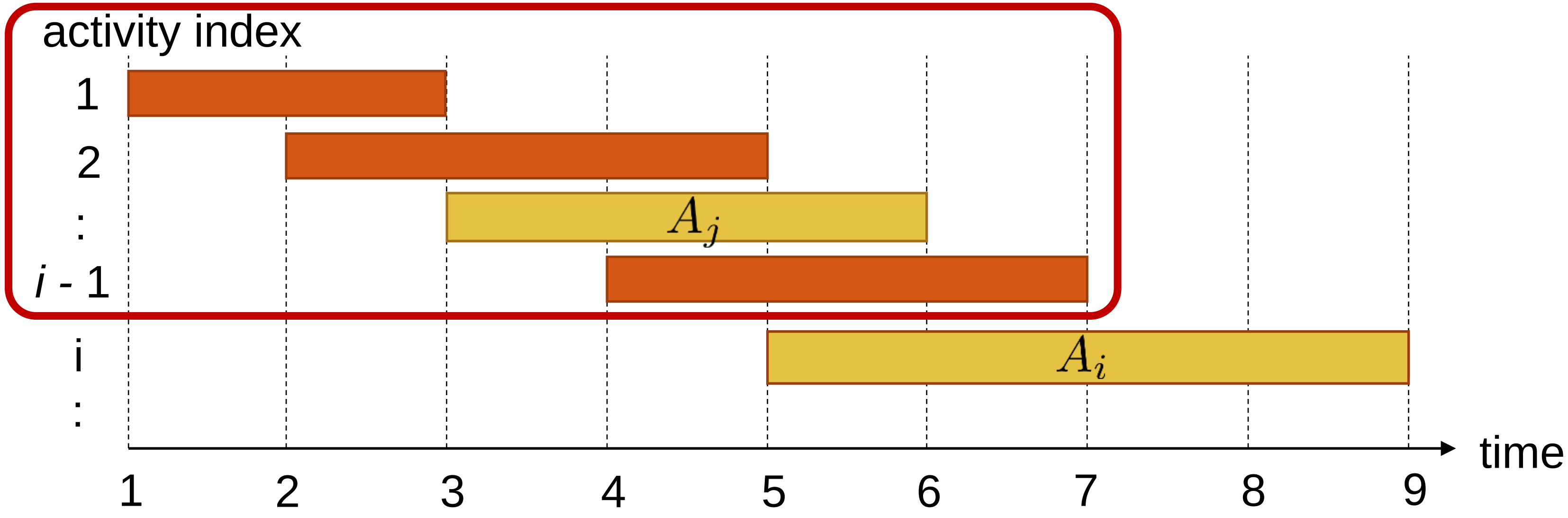
select the i-th activity

Why does the i-th activity must appear in an OPT?

Greedy-Choice Property

- Goal: $1 + M_{p(i)} \geq M_{i-1}$
- Proof
  - Assume there is an OPT solution for the first $i - 1$ activities (M_{i-1})
    - A_j is the last activity in the OPT solution → $M_{i-1} = 1 + M_{p(j)}$
  - Replacing A_j with A_i does not make the OPT worse

$$1 + M_{p(i)} \geq 1 + M_{p(j)} = M_{i-1}$$



Pseudo Code

Activity-Selection Problem
Input: n activities with $\langle s_i, f_i \rangle$, $p(j)$ = largest index $i < j$ s.t. i and j are compatible
Output: the maximum number of activities

```
Act-Select(n, s, f, v, p)
  M[0] = 0
  for i = 1 to n
      M[i] = 1 + M[p[i]]
  return M[n]
```

$$T(n) = \Theta(n)$$

```
Find-Solution(M, n)
  if n = 0
    return {}
  return {n} ∪ Find-Solution(p[n])
```

$$T(n) = \Theta(n)$$

Select the **last** compatible one (←) = Select the **first** compatible one (→)

Greedy #2: Coin Changing

Textbook Exercise 16.1

Coin Changing Problem

- Input: n dollars and unlimited coins with values $\{v_i\}$ (1, 5, 10, 50)
- Output: the minimum number of coins with the total value n

- **Cashier's algorithm**: at each iteration, add the coin with the largest value no more than the current total

Does this algorithm return the OPT?

Step 1: Cast Optimization Problem

Coin Changing Problem
Input: n dollars and unlimited coins with values $\{v_i\}$ (1, 5, 10, 50)
Output: the minimum number of coins with the total value n

- Subproblems
  - `C(i)`: minimal number of coins for the total value i
  - Goal: `C(n)`

Step 2: Prove Optimal Substructure

Coin Changing Problem
Input: n dollars and unlimited coins with values $\{v_i\}$ (1, 5, 10, 50)
Output: the minimum number of coins with the total value n

- Suppose OPT is an optimal solution to `C(i)`, there are 4 cases:
  - Case 1: coin 1 in OPT
    - OPT\coin1 is an optimal solution of `C(i - v_1)`
  - Case 2: coin 2 in OPT
    - OPT\coin2 is an optimal solution of `C(i - v_2)`
  - Case 3: coin 3 in OPT
    - OPT\coin3 is an optimal solution of `C(i - v_3)`
  - Case 4: coin 4 in OPT
    - OPT\coin4 is an optimal solution of `C(i - v_4)`

$$C_i = \min_j (1 + C_{i - v_j})$$

Step 3: Prove Greedy-Choice Property

Coin Changing Problem
Input: n dollars and unlimited coins with values $\{v_i\}$ (1, 5, 10, 50)
Output: the minimum number of coins with the total value n

- Greedy choice: select the coin with the largest value no more than the current total
- Proof via contradiction (use the case $10 \leq i < 50$ for demo)
  - Assume that there is no OPT including this greedy choice (choose 10)

    → all OPT use 1, 5, 50 to pay i
    - 50 cannot be used
    - #coins with value 5 < 2 → otherwise we can use a 10 to have a better output
    - #coins with value 1 < 5 → otherwise we can use a 5 to have a better output
  - We cannot pay i with the constraints (at most $5 + 4 = 9$)

Greedy #3:
Fractional Knapsack Problem

Textbook Exercise 16.2-2

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the <u>maximum value</u> for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - **Fractional Knapsack Problem: 物品可以只拿部分**

Fractional Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W, where we can take **any fraction of items**

- Greedy algorithm: at each iteration, choose the item with the highest $\frac{v_i}{w_i}$ and continue when $W - w_i > 0$

Step 1: Cast Optimization Problem

Fractional Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where we can take **any fraction of items**

- Subproblems
  - `F-KP(i, w)`: fractional knapsack problem within w capacity for the first i items
  - Goal: `F-KP(n, W)`

Step 2: Prove Optimal Substructure

Fractional Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where we can take **any fraction of items**

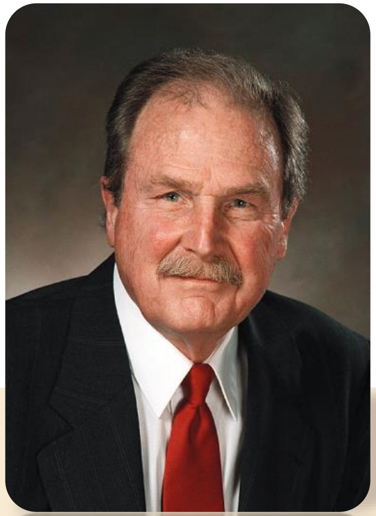
- Suppose OPT is an optimal solution to `F-KP(i, w)`, there are 2 cases:
  - Case 1: full/partial item i in OPT
    - Remove w' of item i from OPT is an optimal solution of `F-KP(i - 1, w – w')`
  - Case 2: item i not in OPT
    - OPT is an optimal solution of `F-KP(i - 1, w)`

Step 3: Prove Greedy-Choice Property

Fractional Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where we can take **any fraction of items**

- Greedy choice: select the item with the highest $\frac{v_i}{w_i}$
- Proof via contradiction ($j = \underset{i}{\operatorname{argmax}} \frac{v_i}{w_i}$)
  - Assume that there is no OPT including this greedy choice
    - If $W \leq w_j$, we can replace all items in OPT with item j
    - If $W > w_j$, we can replace any item weighting w_j in OPT with item j
  - The total value must be equal or higher, because item j has the highest $\frac{v_i}{w_i}$

Do other knapsack problems have this property?

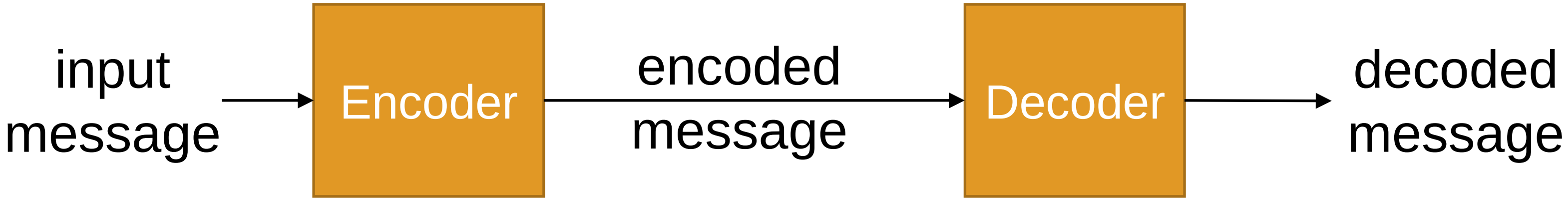


Greedy #4: Huffman Codes

Textbook Chapter 16.3 – Huffman codes

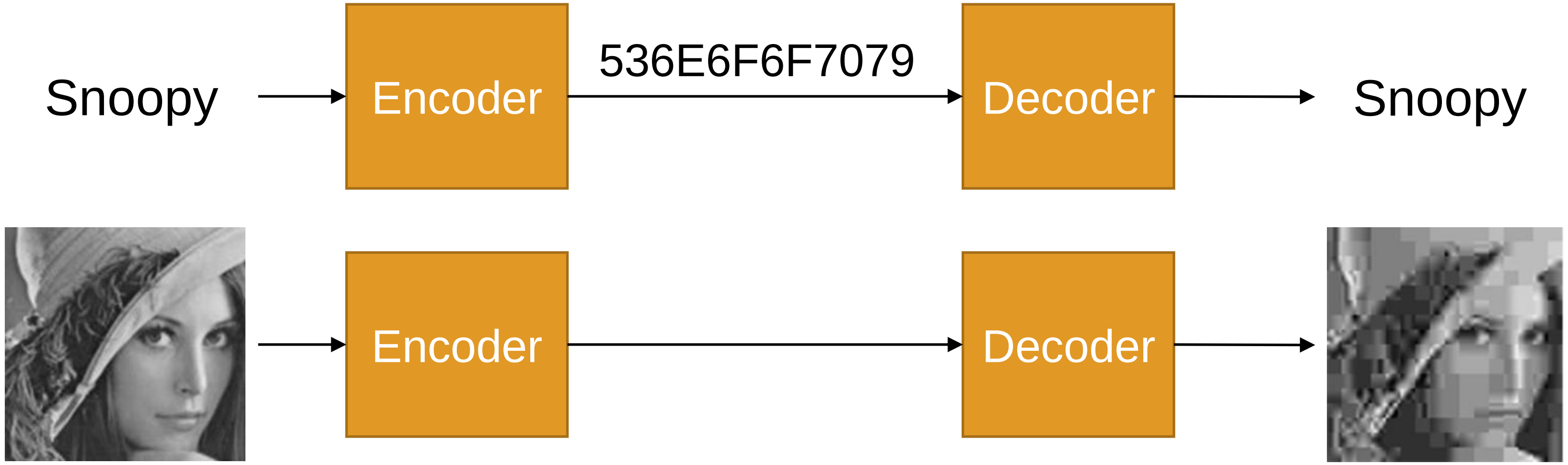
Encoding & Decoding

- **Code (編碼)** is a 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.

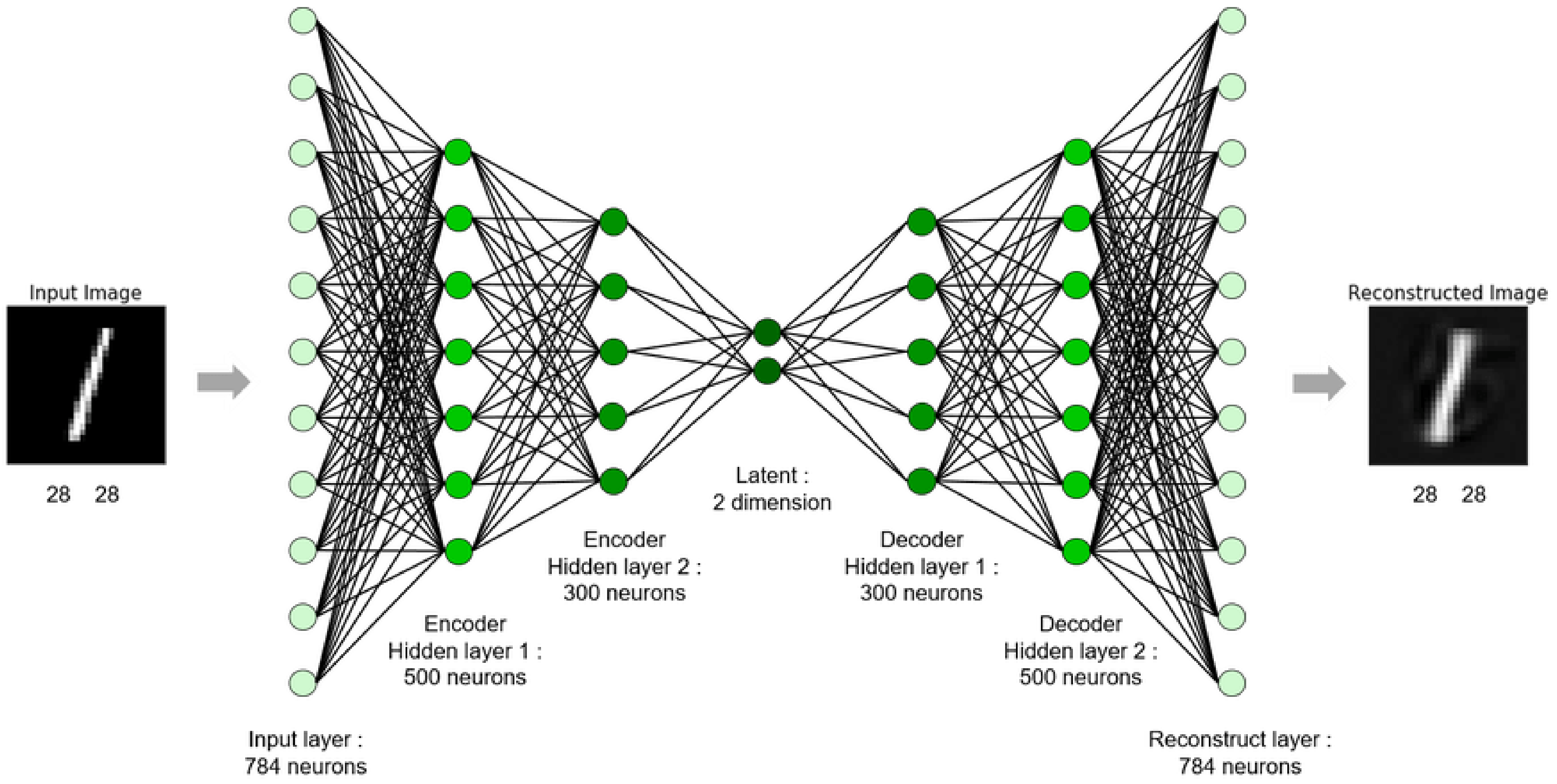


Encoding & Decoding

- Goal
  - Enable communication and storage
  - Detect or correct errors introduced during transmission
  - Compress data: lossy or lossless



Lossy Data Compression: Autoencoder



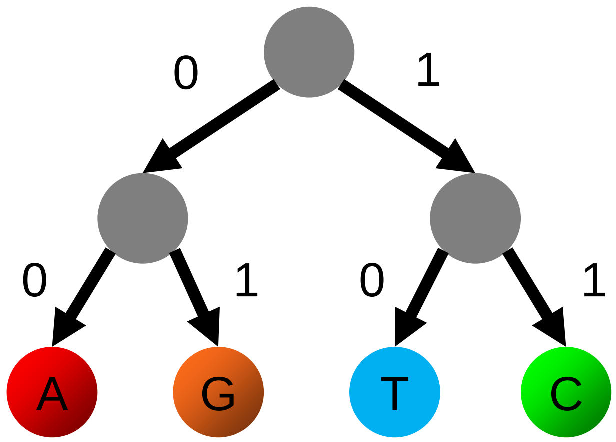
Lossless Data Compression

- Goal: encode each symbol using a unique binary code (w/o ambiguity)
  - How to represent symbols?
  - How to ensure decode(encode(x))=x?
  - How to minimize the number of bits?

Lossless Data Compression

- Goal: encode each symbol using a unique binary code (w/o ambiguity)
  - **How to represent symbols?**
  - How to ensure decode(encode(x))=x?
  - How to minimize the number of bits?

find a binary tree



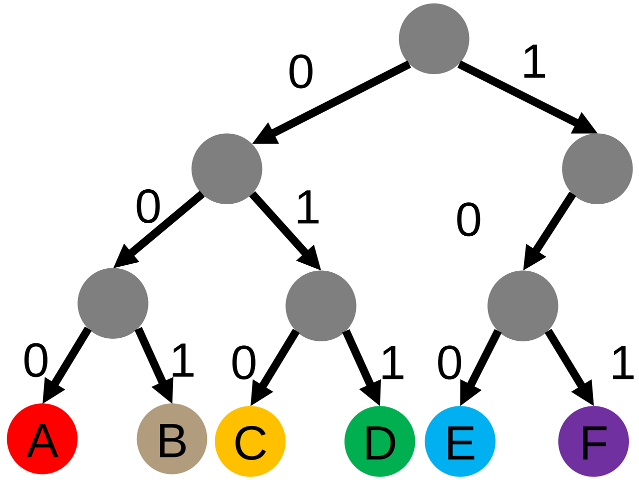
10101101011010100101010010

T T C G G T T T G G G A T

Code

Symbol	A	B	C	D	E	F
Frequency (K)	*45*	*13*	*12*	*16*	*9*	*5*
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

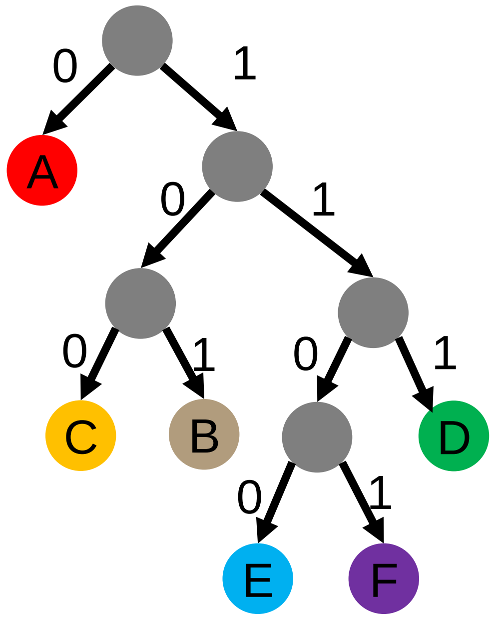
- **Fixed-length**: use the same number of bits for encoding every symbol
  - Ex. ASCII, Big5, UTF



- The length of this sequence is

$$(45 + 13 + 12 + 16 + 9 + 5) \cdot 3 = 300$$

- **Variable-length**: shorter codewords for more frequent symbols



- The length of this sequence is

$$45 \cdot 1 + (13 + 12 + 16) \cdot 3 + (9 + 5) \cdot 4 = 224$$

Lossless Data Compression

- Goal: encode each symbol using a unique binary code (w/o ambiguity)
  - How to represent symbols?
  - **How to ensure decode(encode(x))=x?**
  - How to minimize the number of bits?

use codes that are uniquely decodable

Prefix Code

- Definition: a variable-length code where no codeword is a prefix of some other codeword

Symbol		A	B	C	D	E	F
Frequency (K)		*45*	*13*	*12*	*16*	*9*	*5*
Variable-length	Prefix code	0	101	100	111	1101	1100
	Not prefix code	0	**101**	**10**	111	1101	1100

- Ambiguity: decode(1011100) can be ‘BF’ or ‘CDAA’

prefix codes are uniquely decodable

Lossless Data Compression

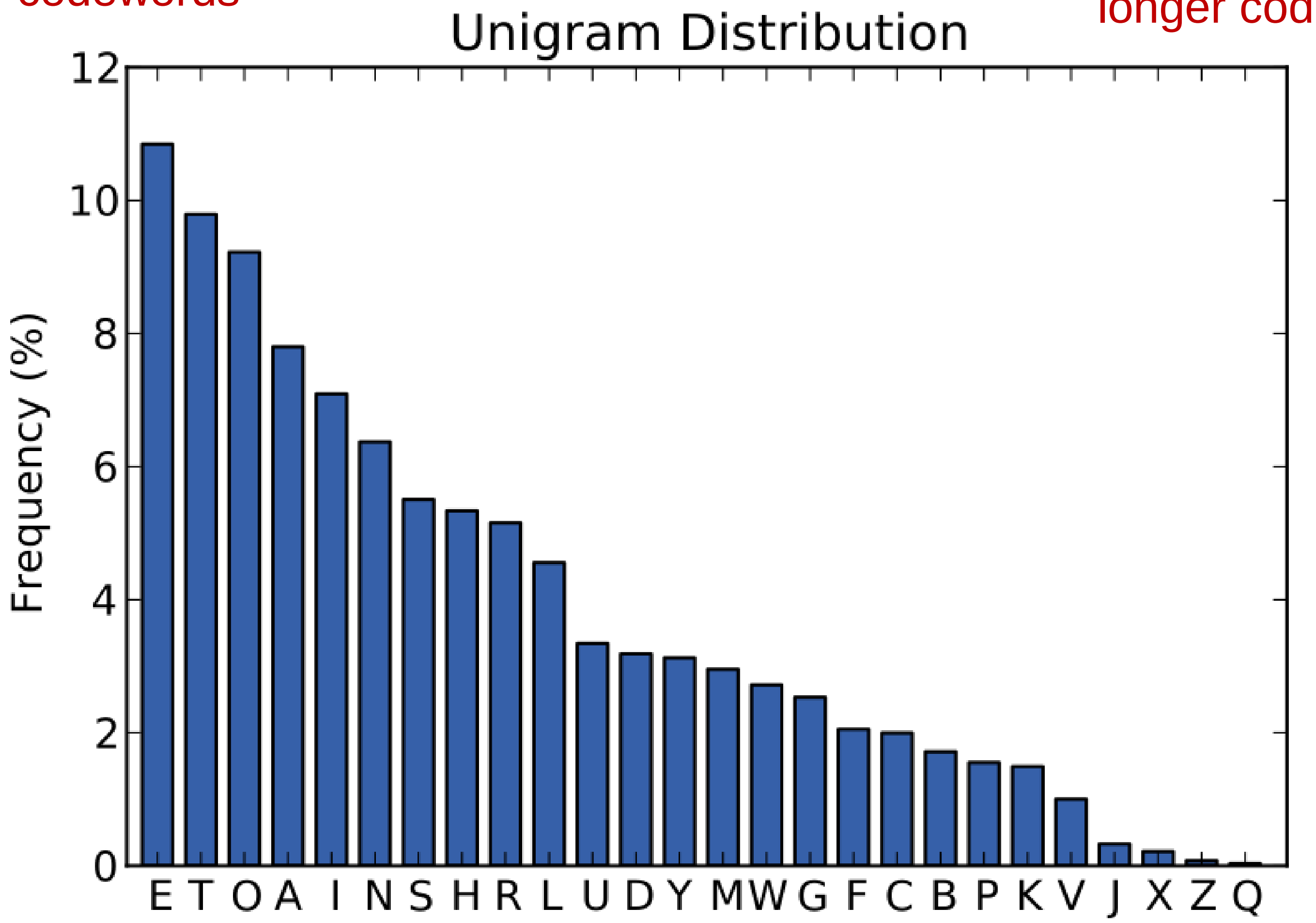
- Goal: encode each symbol using a unique binary code (w/o ambiguity)
  - How to represent symbols?
  - How to ensure decode(encode(x))=x?
  - **How to minimize the number of bits?**

more frequent symbols should use shorter codewords

Letter Frequency Distribution

shorter codewords

longer codewords



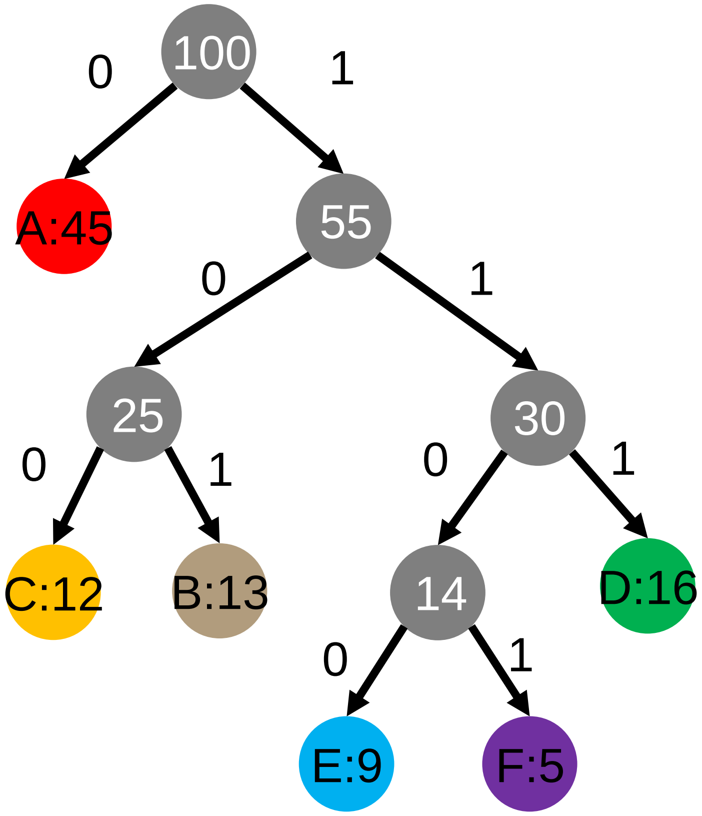
Total Length of Codes

- The weighted depth of a leaf = weight of a leaf (freq) × depth of a leaf
- Total length of codes = Total weighted depth of leaves
- Cost of the tree T

$$B(T) = \sum_{c \in C} \text{freq}(c) \cdot d_T(c)$$

- Average bits per character

$$\frac{B(T)}{100} = \sum_{c \in C} \text{relative-freq}(c) \cdot d_T(c)$$



How to find the **optimal prefix code** to **minimize the cost**?

Prefix Code Problem

- Input: n positive integers $w_1, w_2, \dots, w_n$ indicating word frequency
- Output: a binary tree of n leaves, whose weights form $w_1, w_2, \dots, w_n$ s.t. the cost of the tree is minimized

$$T^* = \arg\min_T B(T) = \arg\min_T \sum_{c \in C} \text{freq}(c) \cdot d_T(c)$$

Step 1: Cast Optimization Problem

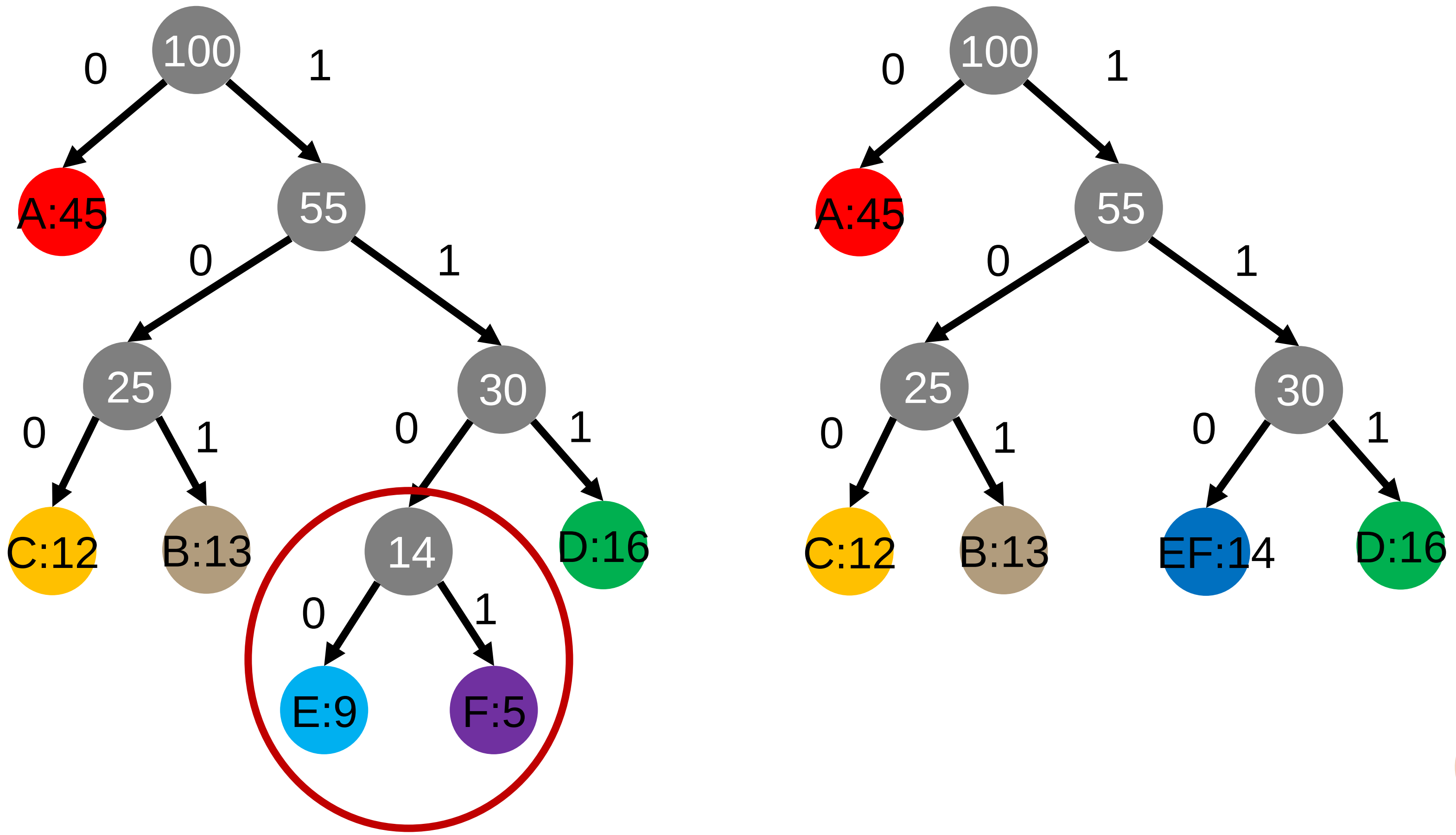
Prefix Code Problem
Input: n positive integers $w_1, w_2, \ldots, w_n$ indicating word frequency
Output: a binary tree of n leaves with minimal cost

- Subproblem: merge two characters into a new one whose weight is their sum
  - `PC(i)`: prefix code problem for i leaves
  - Goal: `PC(n)`

`PC(n)` → `PC(n - 1)`

- Issues
  - It is not the subproblem of the original problem
  - The cost of two merged characters should be considered

Example

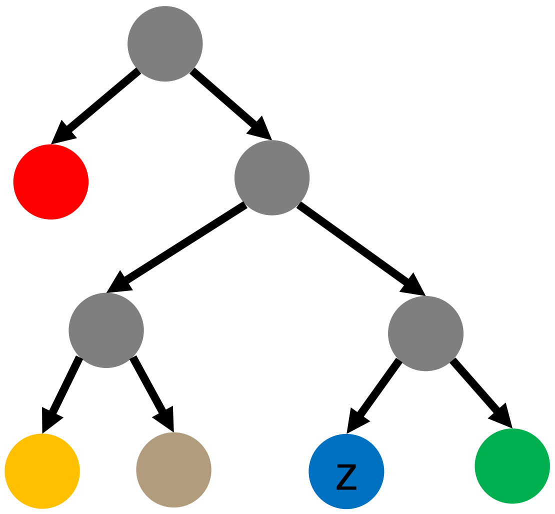

100
0
1
A:45
55
0
1
25
30
0
1
0
1
C:12
B:13
14
D:16
0
1
E:9
F:5
100
0
1
A:45
55
0
1
25
30
0
1
0
1
C:12
B:13
EF:14
D:16

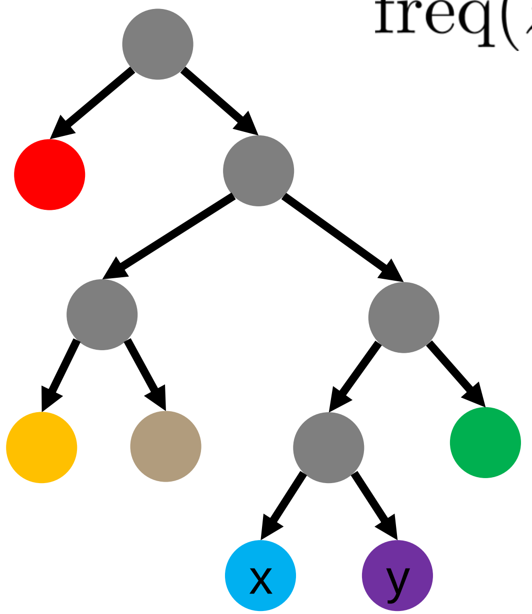
Step 2: Prove Optimal Substructure

Prefix Code Problem
Input: n positive integers $w_1, w_2, \ldots, w_n$ indicating word frequency
Output: a binary tree of n leaves with minimal cost

- Suppose T' is an optimal solution to `PC(i, {w_{1…i-1}, z})`
- T is an optimal solution to `PC(i+1, {w_{1…i-1}, x, y})`

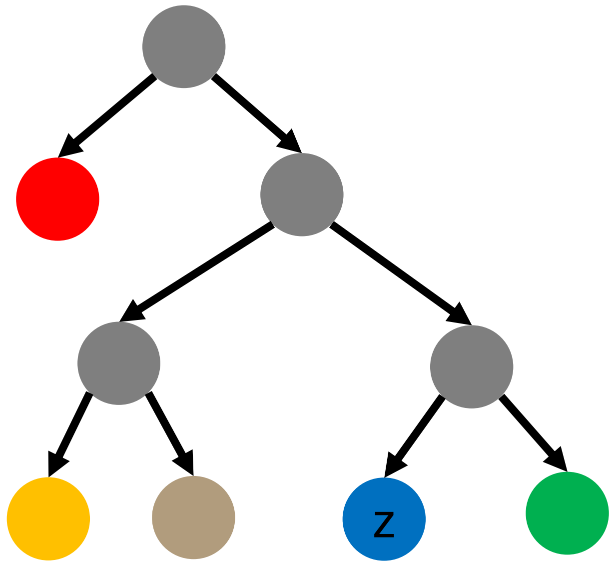
$$\text{freq}(z) = \text{freq}(x) + \text{freq}(y)$$



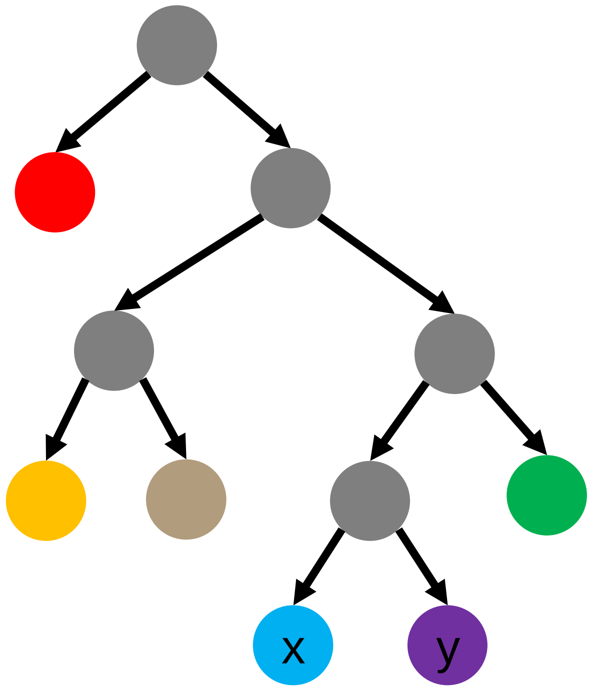


Step 2: Prove Optimal Substructure

- T'



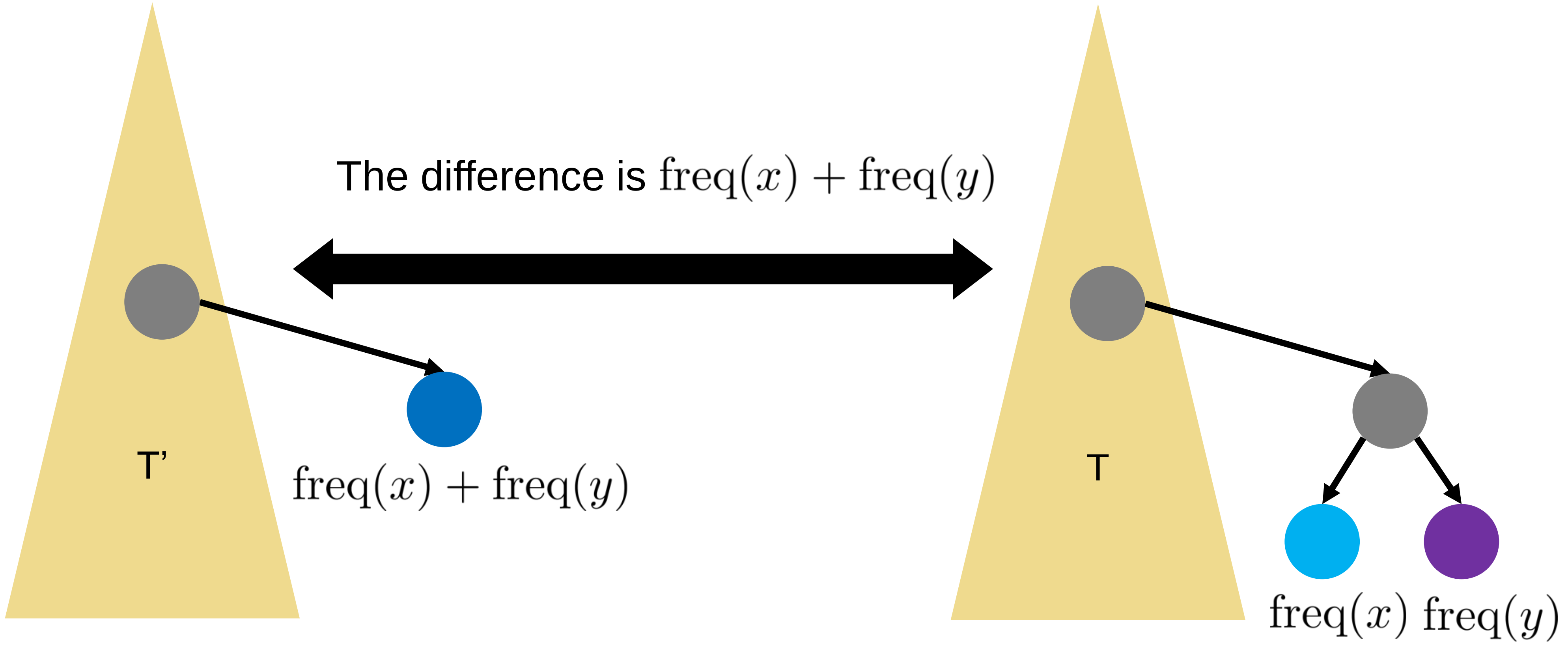
- T



$$
\begin{aligned}
B(T) &= B(T') - \text{freq}(z)d_{T'}(z) + \text{freq}(x)d_T(x) + \text{freq}(y)d_T(y) \\
&= B(T') - (\text{freq}(x) + \text{freq}(y))d_{T'}(z) + \text{freq}(x)(1 + d_{T'}(z)) + \text{freq}(y)(1 + d_{T'}(z)) \\
&= B(T') + \text{freq}(x) + \text{freq}(y)
\end{aligned}
$$

Step 2: Prove Optimal Substructure

- Optimal substructure: T' is OPT if and only if T is OPT

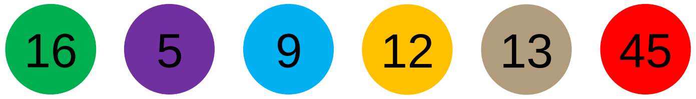


Greedy Algorithm Design

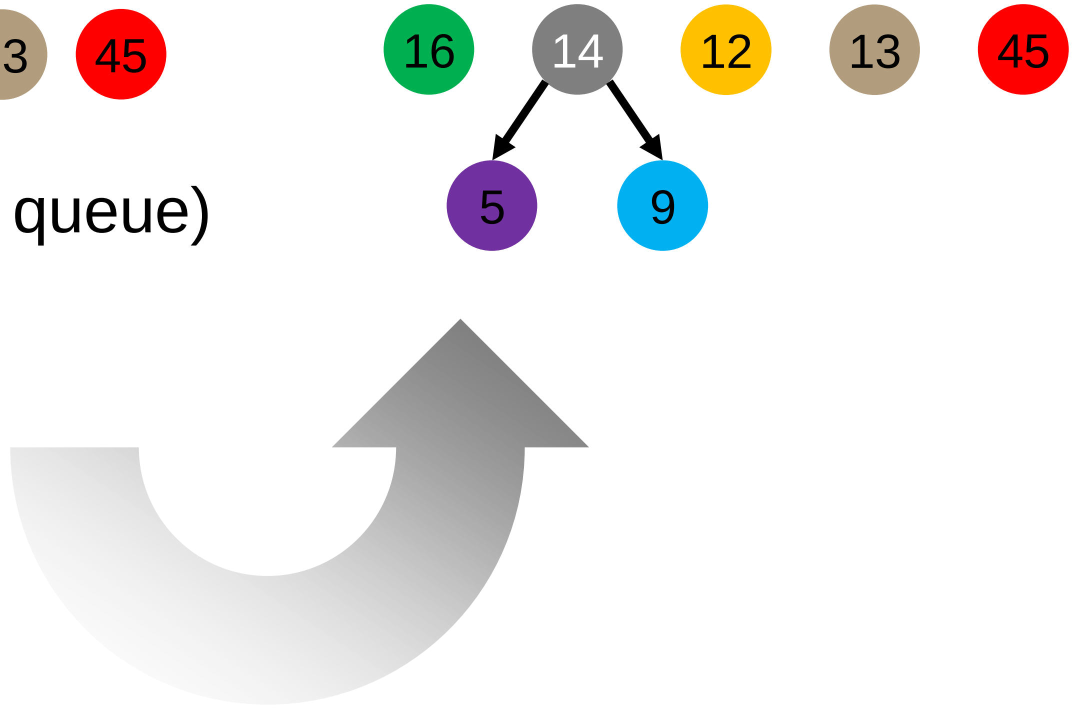
Prefix Code Problem
Input: n positive integers $w_1, w_2, \ldots, w_n$ indicating word frequency
Output: a binary tree of n leaves with minimal cost

- Greedy choice: merge repeatedly until one tree left
  - Select two trees x, y with minimal frequency roots $\mathrm{freq}(x)$ and $\mathrm{freq}(y)$
  - Merge into a single tree by adding root z with the frequency $\mathrm{freq}(x) + \mathrm{freq}(y)$

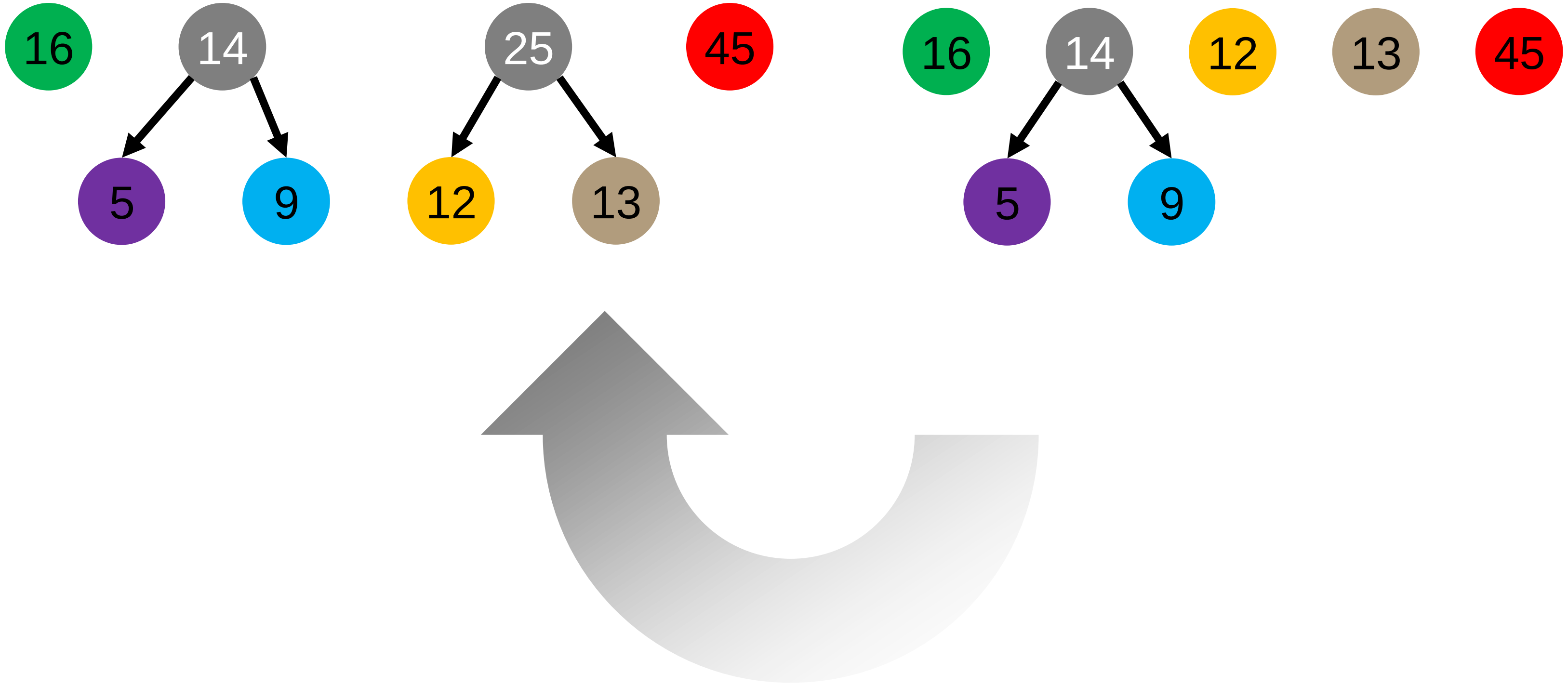
Example



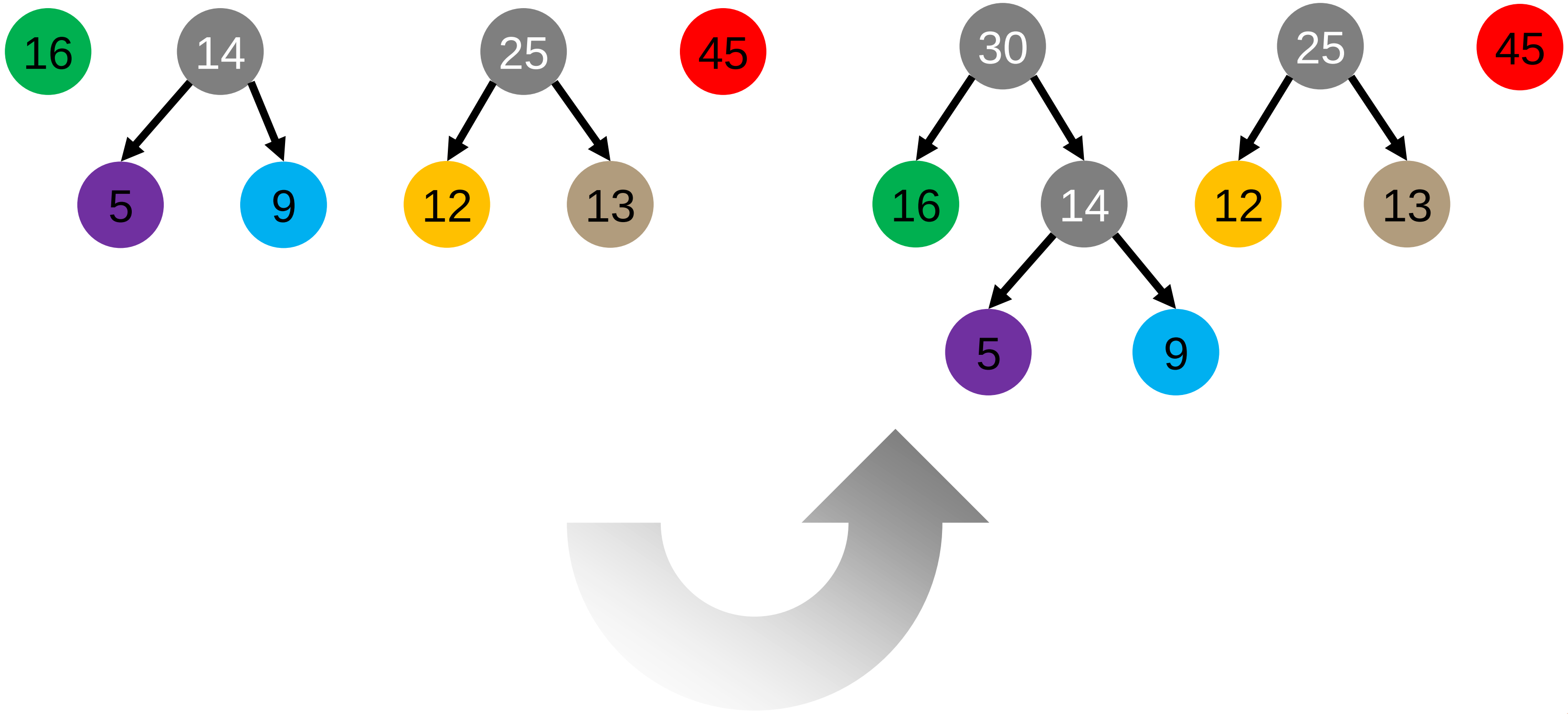
Initial set (store in a priority queue)



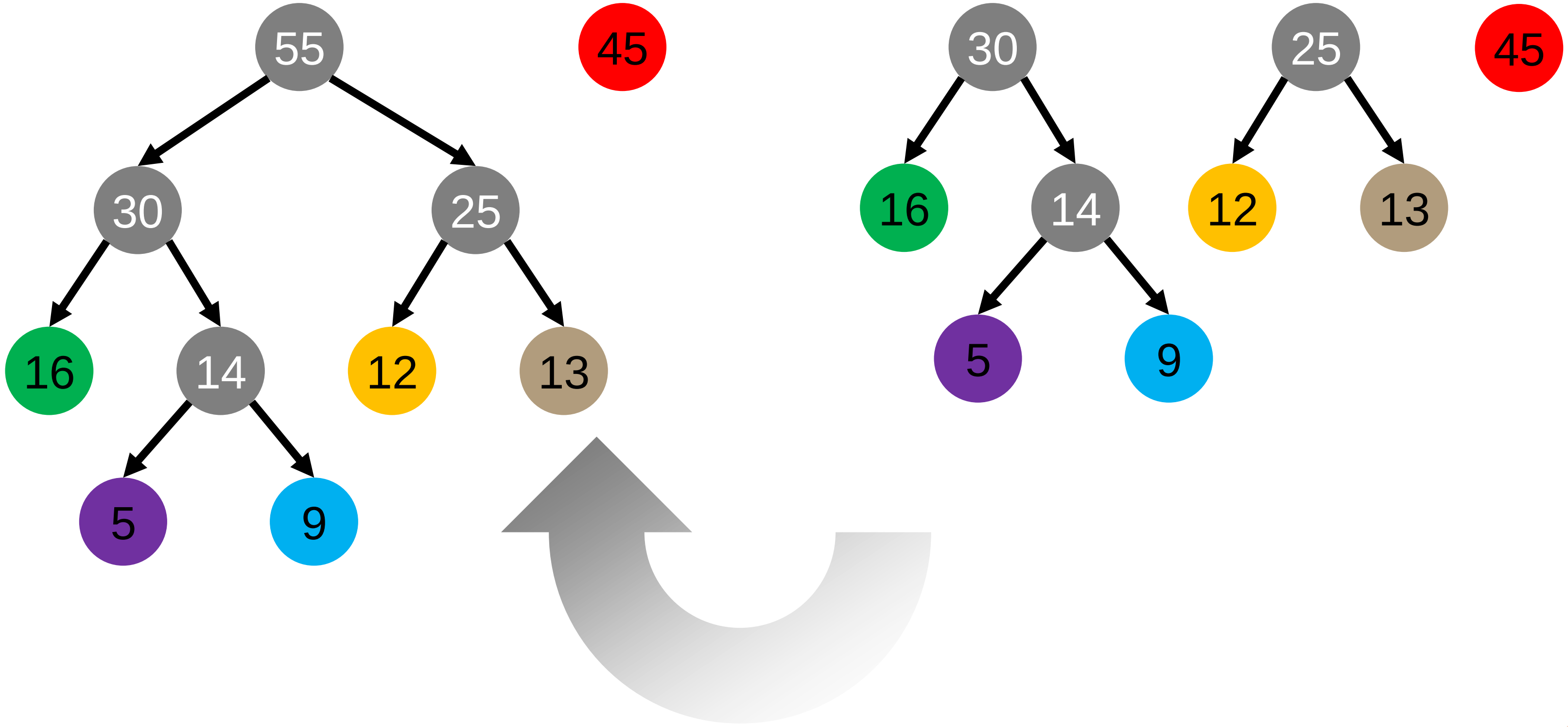
Example



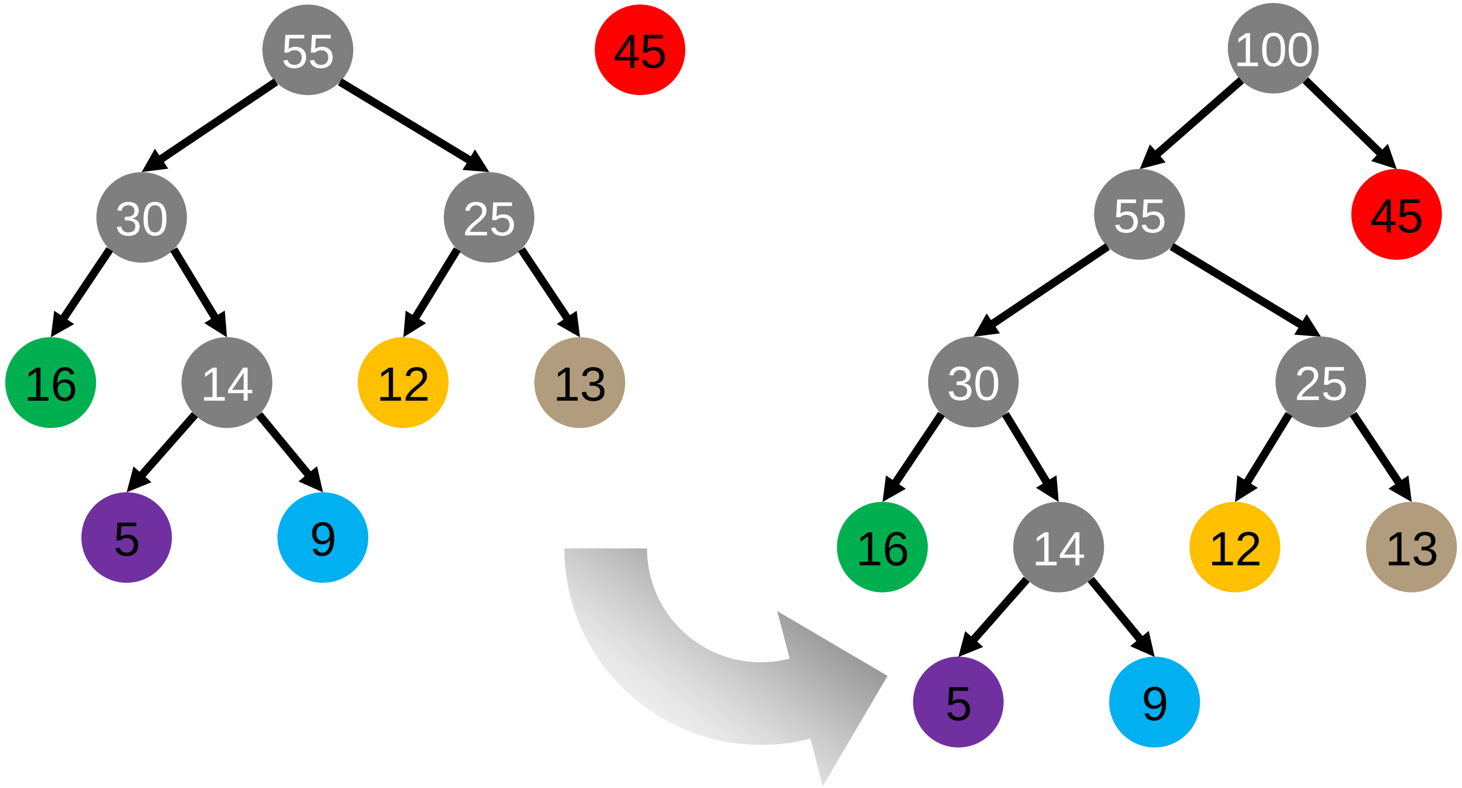
Example



Example



Example

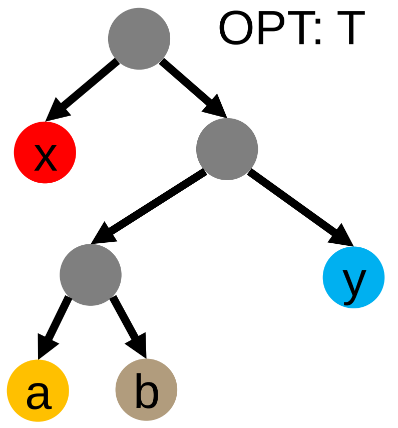


Step 3: Prove Greedy-Choice Property

Prefix Code Problem
Input: n positive integers $w_1, w_2, \dots, w_n$ indicating word frequency
Output: a binary tree of n leaves with minimal cost

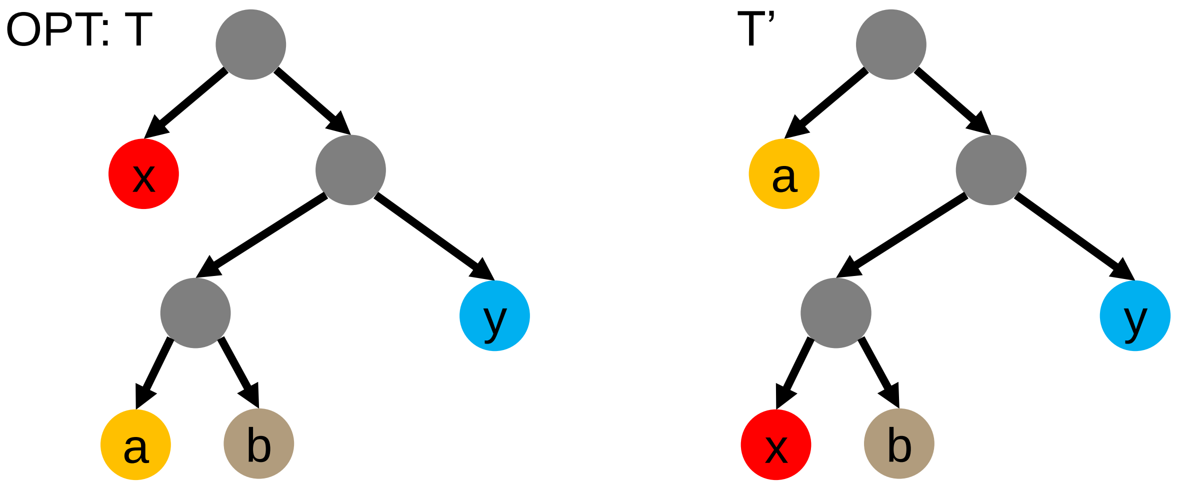
- Greedy choice: merge two nodes with min weights repeatedly
- Proof via contradiction
  - Assume that there is no OPT including this greedy choice
    - x and y are two symbols with lowest frequencies
    - a and b are siblings with largest depths
    - WLOG, assume $\text{freq}(a) \le \text{freq}(b)$ and $\text{freq}(x) \le \text{freq}(y)$

→ $\text{freq}(x) \le \text{freq}(a)$ and $\text{freq}(y) \le \text{freq}(b)$



  - Exchanging a with x and then b with y can make the tree equally or better

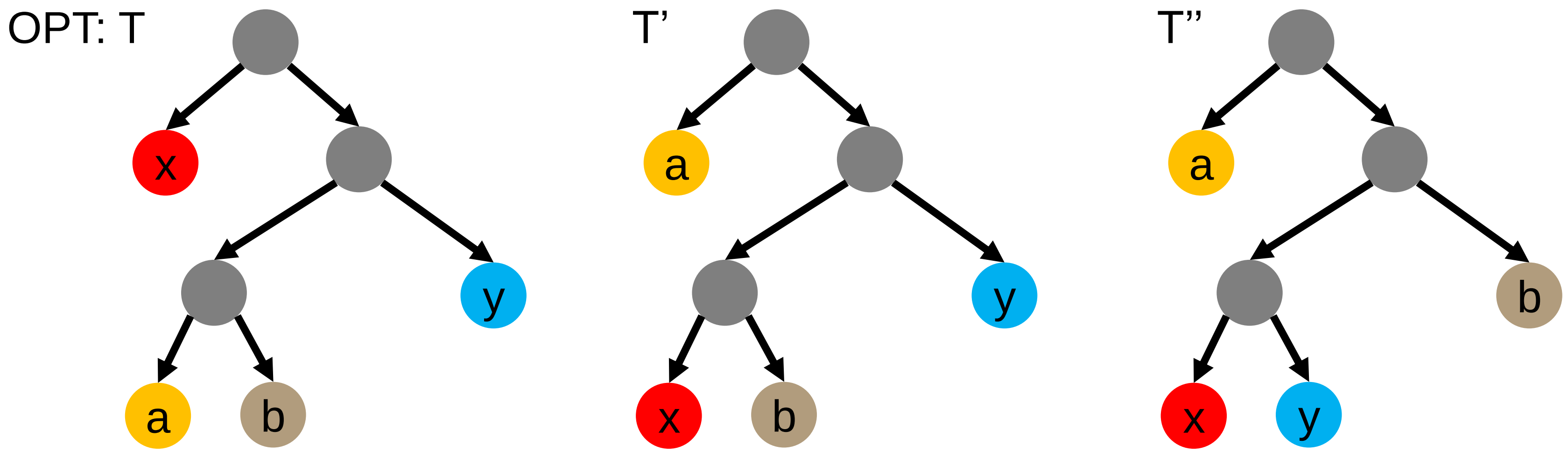
Step 3: Prove Greedy-Choice Property



$$
\begin{aligned}
B(T) - B(T') &= \textstyle\sum_{s \in S} \mathrm{freq}(s) d_T(s) - \sum_{s \in S} \mathrm{freq}(s) d_{T'}(s) \\
&= \mathrm{freq}(x) d_T(x) + \mathrm{freq}(a) d_T(a) - \mathrm{freq}(x) d_{T'}(x) - \mathrm{freq}(a) d_{T'}(a) \\
&= \mathrm{freq}(x) d_T(x) + \mathrm{freq}(a) d_T(a) - \mathrm{freq}(x) d_T(a) - \mathrm{freq}(a) d_T(x) \\
&= (\mathrm{freq}(a) - \mathrm{freq}(x))(d_T(a) - d_T(x)) \;\geq 0 \quad \because \mathrm{freq}(x) \leq \mathrm{freq}(a)
\end{aligned}
$$

- Because T is OPT, T’ must be another optimal solution.

Step 3: Prove Greedy-Choice Property



$$
\begin{aligned}
B(T') - B(T'') &= \textstyle\sum_{s\in S} \text{freq}(s) d_{T'}(s) - \sum_{s \in S} \text{freq}(s) d_{T''}(s) \\
&= \text{freq}(y) d_{T'}(y) + \text{freq}(b) d_{T'}(b) - \text{freq}(y) d_{T''}(y) - \text{freq}(b) d_{T''}(b) \\
&= \text{freq}(y) d_{T'}(y) + \text{freq}(b) d_{T'}(b) - \text{freq}(y) d_{T'}(b) - \text{freq}(b) d_{T'}(y) \\
&= (\text{freq}(b) - \text{freq}(y))(d_{T'}(b) - d_{T'}(y)) \geq 0 \quad \because \text{freq}(y) \leq \text{freq}(b)
\end{aligned}
$$

- Because T' is OPT, T'' must be another optimal solution.

Practice: prove the optimal tree must be a full tree

Correctness and Optimality

- Theorem: Huffman algorithm generates an optimal prefix code
- Proof
  - Use induction to prove: Huffman codes are optimal for n symbols
    - $n = 2$, trivial
    - For a set S with $n + 1$ symbols,

1. Based on the greedy choice property, two symbols with minimum frequencies are siblings in T
2. Construct T' by replacing these two symbols x and y with z s.t. $S' = (S \backslash \{x, y\}) \cup \{z\}$ and $\text{freq}(z) = \text{freq}(x) + \text{freq}(y)$
3. Assume T' is the optimal tree for n symbols by inductive hypothesis
4. Based on the optimal substructure property, we know that when T' is optimal, T is optimal too (case $n + 1$ holds)

This induction proof framework can be applied to prove its ***optimality*** using the **optimal substructure** and the **greedy choice property**.

Pseudo Code

Prefix Code Problem
Input: n positive integers $w_1, w_2, \dots, w_n$ indicating word frequency
Output: a binary tree of n leaves with minimal cost

```
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)
    Delete(Q, x)
    Delete(Q, y)
  return Extract-Min(Q) // return the prefix tree
```

$$T(n) = \Theta(n \log n)$$

Drawbacks of Huffman Codes

- Huffman's algorithm is optimal for a symbol-by-symbol coding with a known input probability distribution
- Huffman's algorithm is sub-optimal when
  - blending among symbols is allowed
  - the probability distribution is unknown
  - symbols are not independent

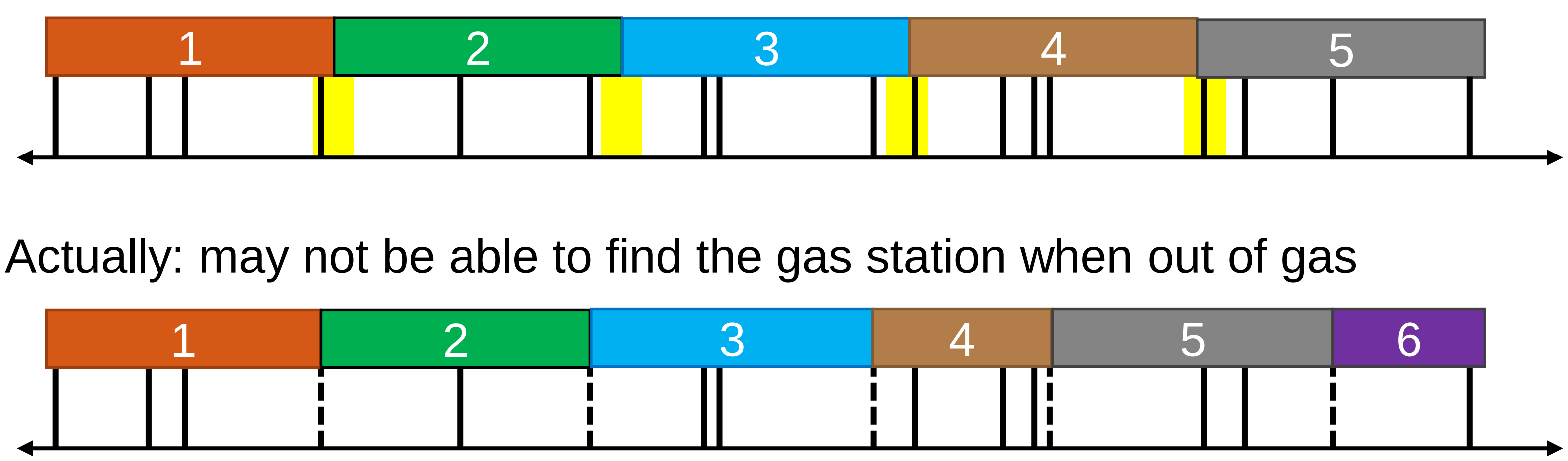
Greedy #5: Breakpoint Selection

Breakpoint Selection Problem

- Input: a planned route with $n+1$ gas stations $b_0, \ldots, b_n$; the car can go at most C after refueling at a breakpoint
- Output: a refueling schedule ($b_0 \rightarrow b_n$) that minimizes the number of stops

Ideally: stop when out of gas

Actually: may not be able to find the gas station when out of gas



- Greedy algorithm: go as far as you can before refueling

Step 1: Cast Optimization Problem

Breakpoint Selection Problem
Input: $n + 1$ breakpoints $b_0, \ldots, b_n$; gas storage is C
Output: a refueling schedule (b_0→b_n) that minimizes the number of stops

- Subproblems
  - `B(i)`: breakpoint selection problem from b_i to b_n
  - Goal: `B(0)`

Step 2: Prove Optimal Substructure

Breakpoint Selection Problem
Input: $n+1$ breakpoints $b_0, \dots, b_n$; gas storage is C
Output: a refueling schedule ($b_0 \rightarrow b_n$) that minimizes the number of stops

- Suppose OPT is an optimal solution to `B(i)` where j is the largest index satisfying $b_j - b_i \leq C$, there are $j - i$ cases
  - Case 1: stop at b_{i+1}
    - OPT+{b_{i+1}} is an optimal solution of `B(i + 1)`
  - Case 2: stop at b_{i+2}
    - OPT+{b_{i+2}} is an optimal solution of `B(i + 2)`

  :
  - Case $j - i$: stop at b_j
    - OPT+{b_j} is an optimal solution of `B(j)`

$$B_i = \min_{i<k\leq j}(1 + B_k)$$

Step 3: Prove Greedy-Choice Property

Breakpoint Selection Problem
Input: $n + 1$ breakpoints $b_0, \ldots, b_n$; gas storage is C
Output: a refueling schedule ($b_0 \rightarrow b_n$) that minimizes the number of stops

- Greedy choice: go as far as you can before refueling (select b_j)
- Proof via contradiction
  - Assume that there is no OPT including this greedy choice (after b_i then stop at b_k, $k \neq j$)
    - If $k > j$, we cannot stop at b_k due to out of gas
    - If $k < j$, we can replace the stop at b_k with the stop at b_j
  - The total value must be equal or higher, because we refuel later ($b_j > b_k$)

$$B_i = \min_{i<k\leq j}(1 + B_k) \Rightarrow B_i = 1 + B_j$$

Pseudo Code

Breakpoint Selection Problem
Input: $n + 1$ breakpoints $b_0, \ldots, b_n$; gas storage is C
Output: a refueling schedule (b_0→b_n) that minimizes the number of stops

```
BP-Select(C, b)
  Sort(b) s.t. b[0] < b[1] < … < b[n]
  p = 0
  S = {0}
  for i = 1 to n - 1
    if b[i + 1] – b[p] > C
      if i == p
        return "no solution"
      A = A ∪ {i}
      p = i
return A
```

$$T(n) = \Theta(n \log n)$$

To Be Continued…

Question?

Important announcement will be sent to @ntu.edu.tw mailbox & post to the course website

Course Website: http://ada.miulab.tw
Email: ada-ta@csie.ntu.edu.tw