CS211FZ: Data Structures and Algorithms II

Greedy Method

Huffman Codes

LECTURER: Chris Roadknight

Chris.Roadknight@gmail.com

Admin

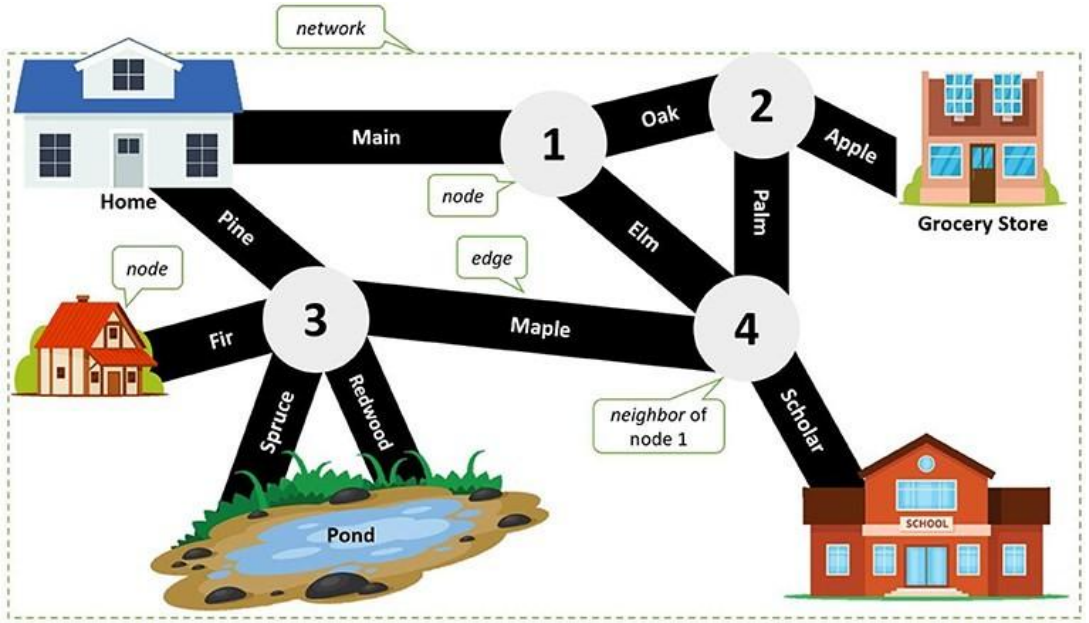
Exam
30th June, 2:30-4:30pm
4 questions on 4 topics

Lab 22nd June.
Quiz (5%)
Lab task (to be completed IN THE LAB) (5%)
I will only accept VERY good reasons for missing this
("it is raining" is not a good enough reason)

Recap

Algorithms for Optimisation Problems

- Algorithms for optimisation problems are designed to find the best solution among a set of possible solutions. Sorting, Searching, Hashing, Ranking
- For example:
  - Finding the shortest route or maximizing profits.



Algorithms for Optimization Problems

- Commonly used algorithms for solving optimization problems include:
  - Dynamic Programing
  - Greedy Algorithms

Bottom Up
Vs
Top Down

Bottom-up vs Top-down

A fundamental decision in many aspects of computer science.

Top-down approaches start with a high-level overview and break down into smaller parts, while bottom-up approaches build up from basic components.

Examples include software development, algorithm design, and parsing, with varying applications in each.

Examples in Different Areas:

Software Development: Top-down might mean designing a university system first, then detailing student and course modules (Waterfall). Bottom-up could involve creating individual classes like "Student" and combining them. (OOP). Both effective but different outcomes…why?

Algorithm Design: For calculating Fibonacci numbers, top-down uses recursion to solve smaller problems first (Divide and conquer), while bottom-up iterates from base cases up (Dynamic programming).

Parsing: Top-down starts with the overall structure of a program and works down, while bottom-up starts with individual tokens and builds up to the full parse tree.

These approaches help tackle complex problems, but their effectiveness depends on the context, like project size or problem complexity.

Top-Down Approach

This method begins with a high-level overview of the system or problem, focusing on the overall functionality before breaking it down into smaller, more manageable components.

It is often associated with decomposition or divide-and-conquer strategies, emphasising planning and understanding the big picture first.

This approach is particularly useful for complex systems where the overall structure needs to be clear before implementation details are addressed.

Because it doesn’t solve subproblems first it can sometimes ‘fail slowly’

Bottom-Up Approach

In contrast, the bottom-up approach starts with the smallest, most basic components and builds up to form the larger system or solution.

It is often used when the fundamental building blocks are well-defined and reusable, and the goal is to construct the complete system by integrating these components.

This method is iterative and can be more flexible, allowing for early testing and validation of individual parts.

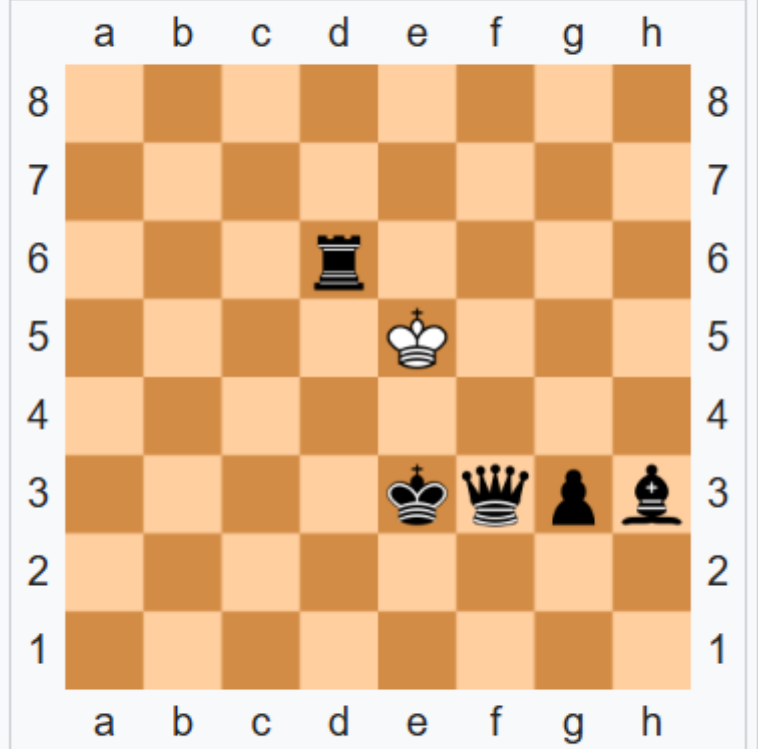
Bottom-up analysis is chess

Retrograde Analysis: This starts from the end of the game, like checkmate or stalemate, and works backwards. It determines the best moves for earlier positions by solving simpler end positions first.

Endgame Tablebases: These are databases for positions with few pieces (e.g., king and pawn vs. king). Bottom-up dynamic programming computes outcomes by starting with positions with no moves left and building up, ensuring every position is evaluated efficiently.

For example, imagine a chess problem where players move a rook to reach a square. Bottom-up dynamic programming would start from the target square, work backwards, and divide the board into winning and losing regions for each player.

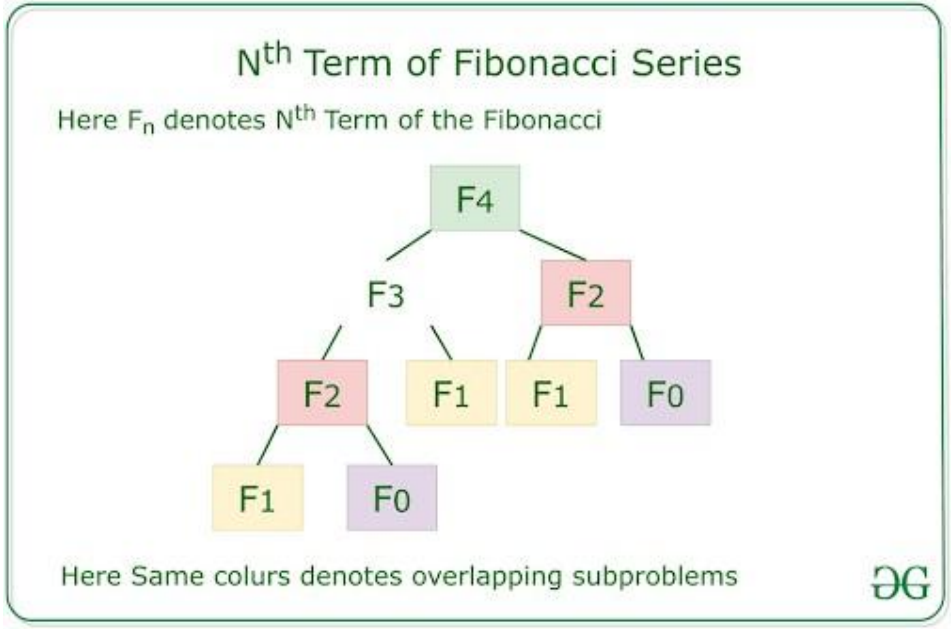
Éric Angelini,
Europe Echecs 433, Apr. 1995



Black to move. What was White's last move?

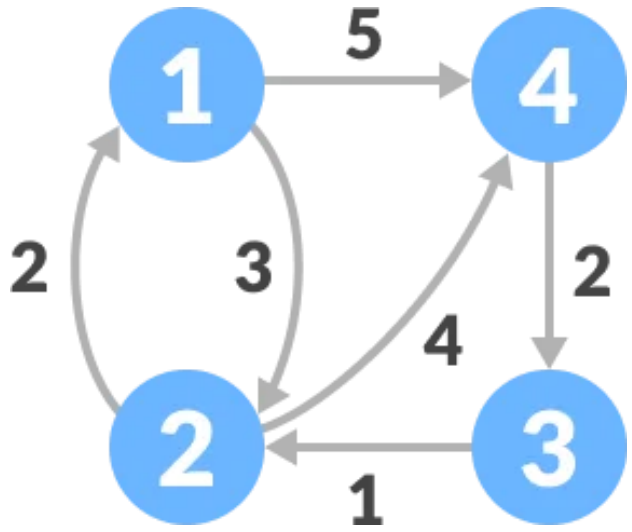
Dynamic Programing

- This approach involves breaking down the main problem into smaller subproblems.
- Solve each subproblem just once, storing the solutions—typically in a table—to avoid redundant work.
- The final solution is built upon optimal solutions to these subproblems.



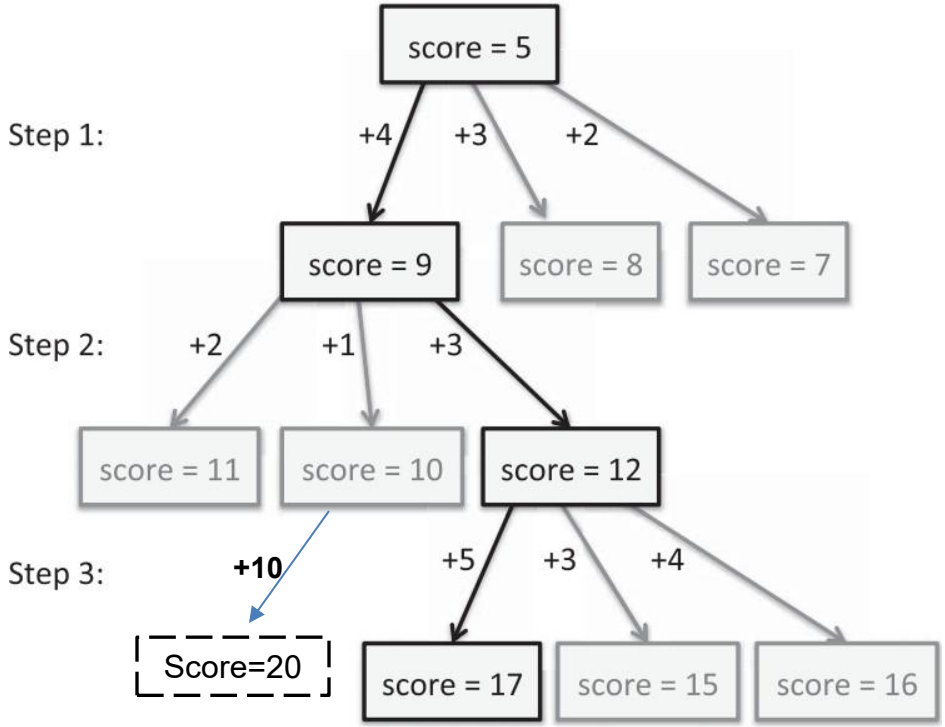
Dynamic Programing

- Examples of dynamic programming:
  - Floyd-Warshall Algorithm: Computes the shortest paths between all pairs of nodes in a graph.
  - Bellman-Ford Algorithm: Computes the shortest paths from a single source node to all other nodes in a graph.
  - Knapsack Problem: Aimed at selecting items with given weights and values to maximize the total value without exceeding a weight limit.



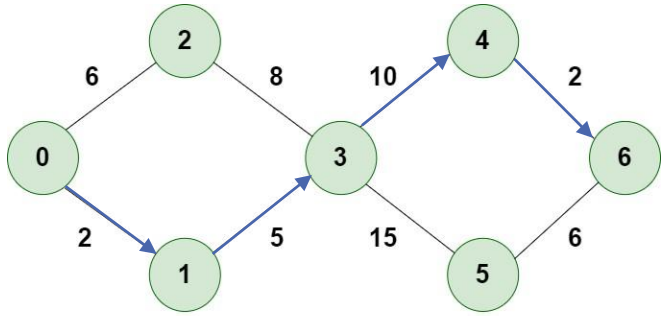
Greedy Algorithm

- Greedy algorithms solve a problem by selecting the best option available at the moment without considering future consequences.
- It doesn't worry whether the current best result will bring the overall optimal result.



Greedy Algorithm

- Example of Greedy Algorithm:
  - Dijkstra's Algorithm: Used to find the shortest path from a starting node to all other nodes in a weighted graph.
  - Prim's Algorithm: Finds the minimum spanning tree of a weighted graph.
  - Huffman Coding: Used in data compression that assigns variable-length codes to input characters, with shorter codes for more frequent characters.



Dijkstra's Algorithm

Dynamic Programming Vs Greedy Algorithm

- Both are employed to find best solution among various possibilities.

Dynamic programing

- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.
- Solve bottom-up. More optimal but slower.

Greedy algorithms

- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Make the choice before solving the subproblems.
- Solve top-down. Less optimal but quicker.

Dynamic Programming Vs Greedy Algorithm

- Chess (Dynamic Programming):
  - In the game of Chess, every time we make a decision about a move, we have to also think about the future consequences. Explore all options.

Exhaustive and can only be applied to a limited depth. But positions can be stored and reused.

Dynamic Programming Vs Greedy Algorithm

- Tennis (Greedy Algorithms):
  - In the game of Tennis, our action is based on immediate advantage rather than long-term benefits. (or not?)

Dynamic Programming Vs Greedy Algorithm

Might tempted to generate a dynamic-programming solution to a problem when a greedy solution is enough.

Greedy Algorithm

Dynamic Programming

you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required.

The knapsack problem is a good example to see the difference.

The 0-1 knapsack problem

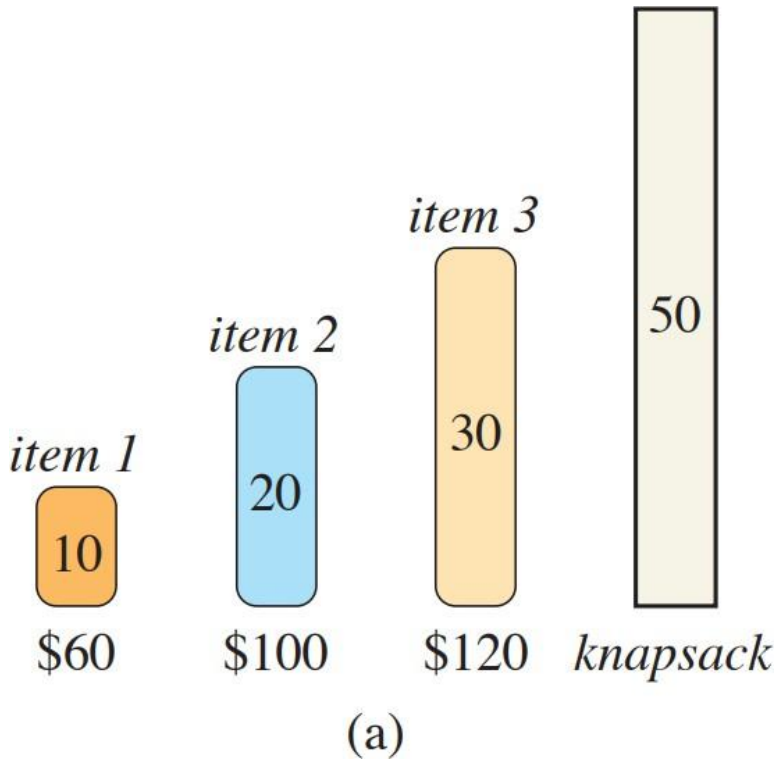
- A person is allowed to take items for free in a knapsack (bag) that has a weight limit of **W** pounds.
- The person can choose to take any subset of n items in the store.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Objective is to find a most valuable subset of items with total weight $\leq W$.
- Must either take an item (1) or not take it (0) —can't take part of it.

The 0-1 knapsack problem

- Suppose we have three items and a knapsack that can hold W = 50 pounds.
  - Item 1 weighs 10 pounds and is worth $60.
  - Item 2 weighs 20 pounds and is worth $100.
  - Item 3 weighs 30 pounds and is worth $120.

- v_i / w_i is a value per pound.

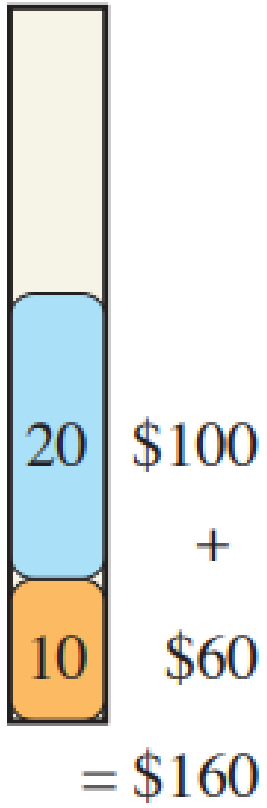
i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4



The 0-1 knapsack problem

- Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.
- **Greedy Solution:**
  - Take the items in order of greatest value per pound.
  - Take items 1 and 2.
  - Value = 160, weight = 30.
  - Have 20 pounds of capacity left over.
  - Suboptimal solution.

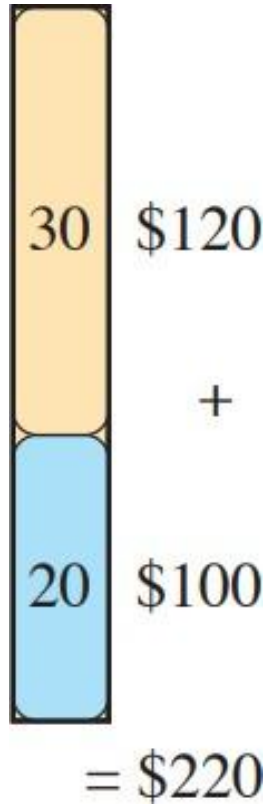
i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4



The 0-1 knapsack problem

- **Optimal solution:**
  - Take items 2 and 3
  - value = 220, weight = 50.
  - No leftover capacity.

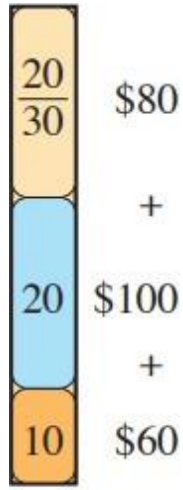
i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4



- Therefore, dynamic programing is best suited for 0-1 knapsack problem. Need to check all the subproblems and then make a choice.

The fractional knapsack problem

- Same as the 0-1 knapsack problem but can take fractions of items, rather than having to make a binary (0-1) choice for each item.
- Both have optimal substructure.
- But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.
- **Greedy Solution:**
  - Take the items in order of greatest value per pound.
  - Take items 1 and 2.
  - Take fraction of item 3 (20 pound out of 30 pound).
  - Value = 240, weight = 50.
  - Optimal solution
  - No leftover capacity.



(c)

Summary of knapsack problems

- The 0-1 knapsack problem does not have greedy choice.
  - Can use dynamic programming to solve the 0-1 problem.
  - Check all the subproblems and then make a choice.

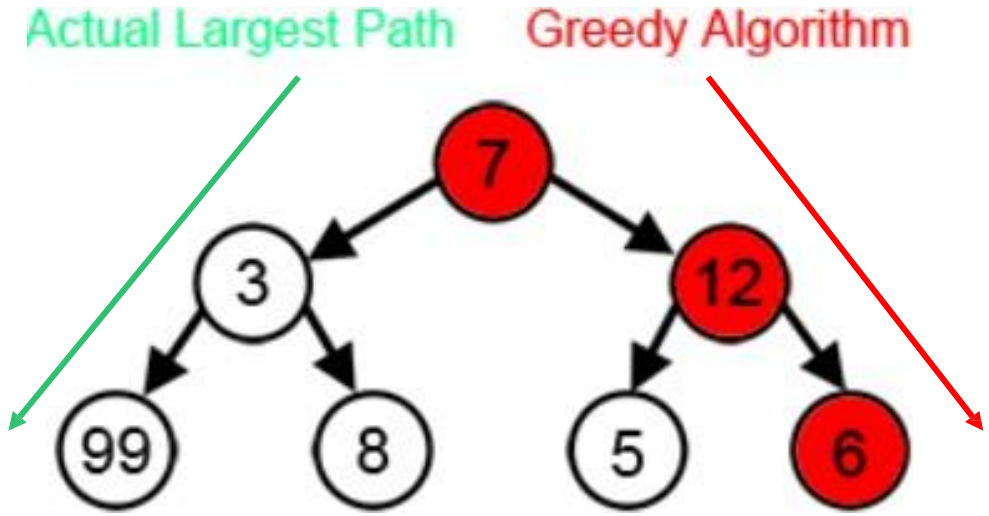
- Fractional knapsack problem has the greedy-choice property.
  - Can use greedy algorithms.
  - Make a choice before solving the subproblems.

Greedy Strategy

- A greedy algorithm finds the best solution to a problem by making choices step by step. The choice that seems best at the moment is the one we go with.

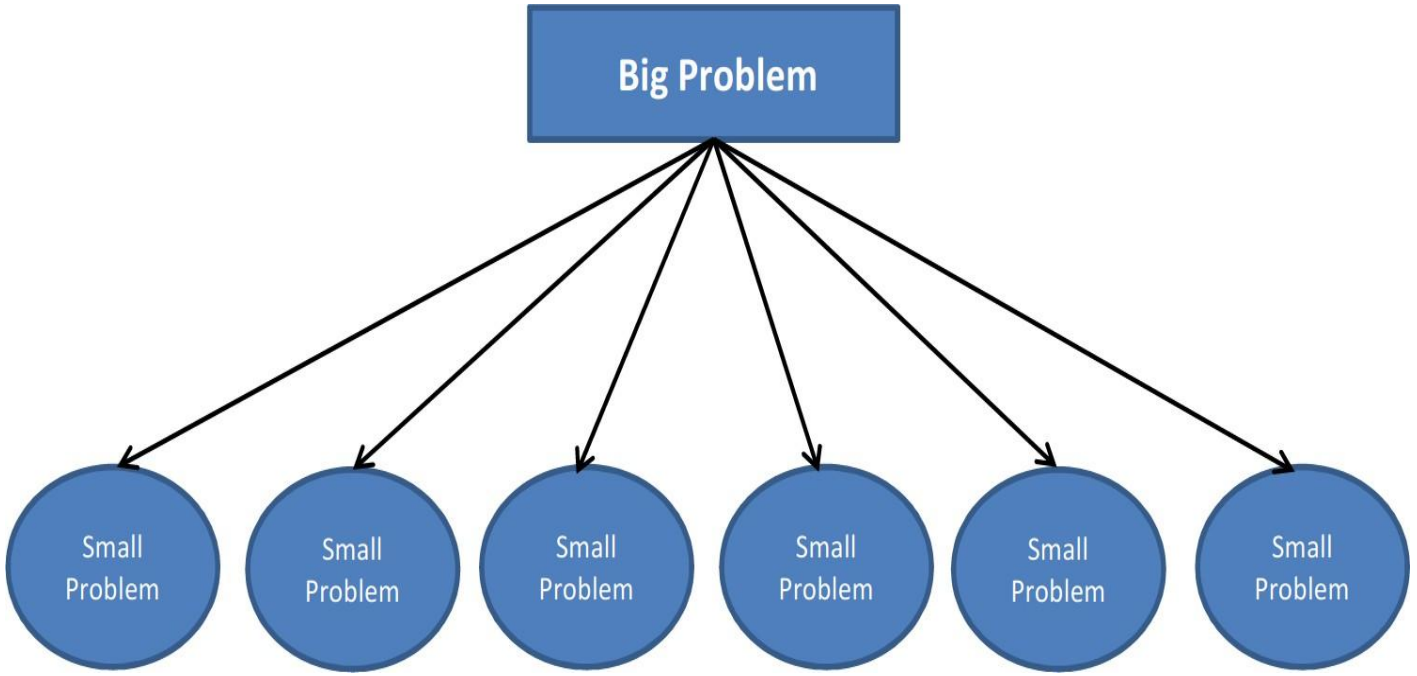
- This approach doesn't always give the absolute best answer, but it often works well.

- Elements of greedy strategy:
  - Divide large problem
  - Recursive solution
  - Make greedy choice
  - Safe greedy choice



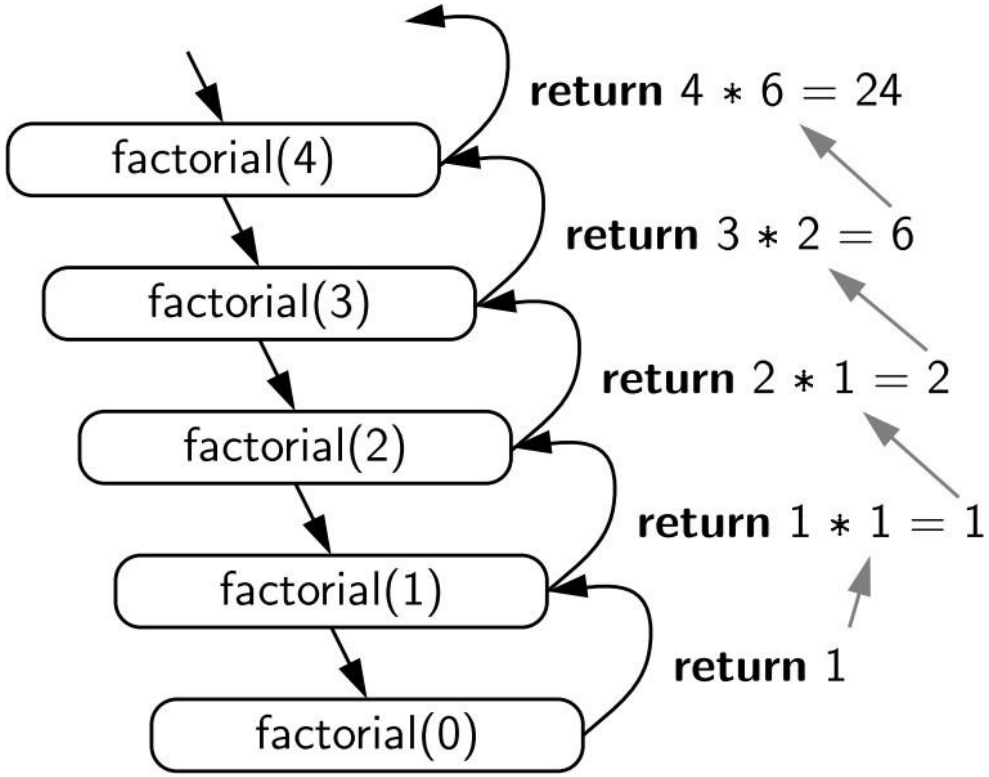
Elements of the greedy strategy - Divide large problem

- Determine the optimal substructure i.e., break down the problem into smaller parts to understand how each choice affects the overall solution.
- For example:
  - Divide and Conquer



Elements of the greedy strategy- Recursive solution

- After dividing big problem into small pieces, a recursive function is defined to that solves each subproblem.
- For example:
  - Calculating Factorial Recursively

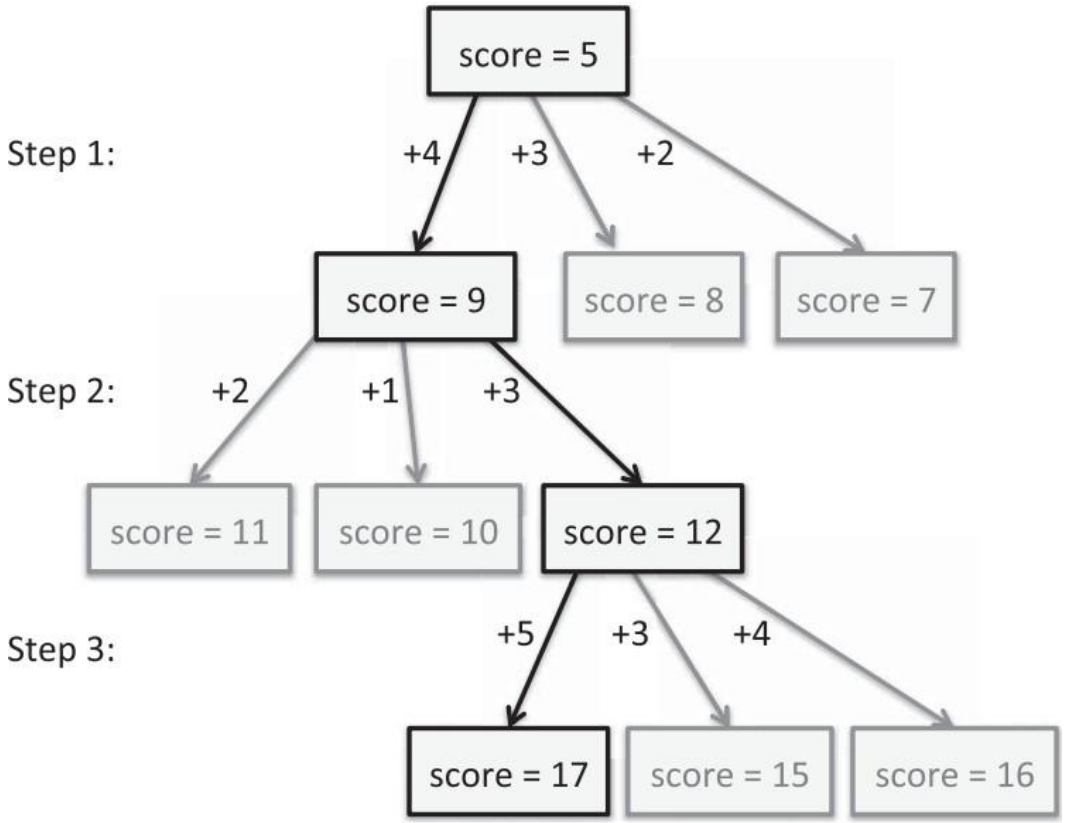


Elements of the greedy strategy- Greedy choice

- Show that if you make the greedy choice, then only one subproblem remains.

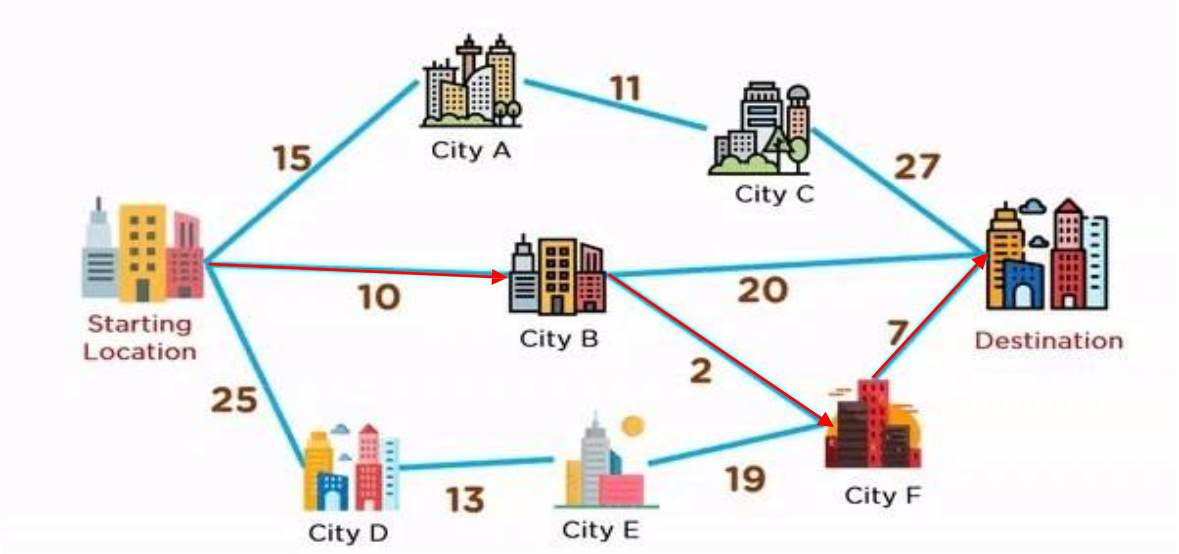
To get a highest score, we make the greedy choice: select a node with highest cost.

After making a greedy choice, out of three available subproblems, we left with only one subproblem.



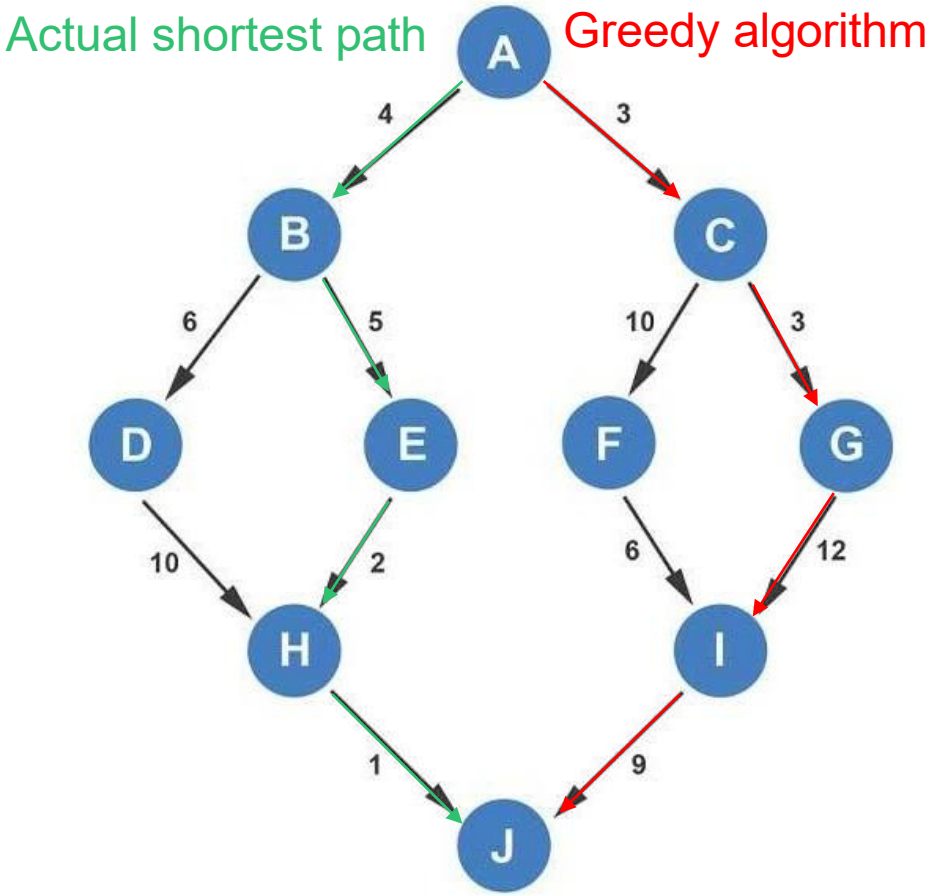
Elements of the greedy strategy - Safe greedy choice

- Ensure that selecting the most favourable option at each stage won't lead to a wrong outcome.
- For example:
  - To find the shortest route on navigation apps under ideal conditions, we select a shortest path but make sure we reach the correct destination.



Elements of the greedy strategy - Safe greedy choice?

- Example 2:
  - Find shortest path to go from A to J.
  - Greedy algorithm will find A-C-G-I-J (27).
  - Actual shortest path is A-B-E-H-J(12).



Not all problems have a safe greedy choice. For example, in the 0/1 Knapsack Problem, choosing the item with the highest value-to-weight ratio at each step doesn't guarantee an optimal solution, so a greedy approach fails.

Huffman codes

Coding

- For storage, each character is represented by some bit sequence: a codeword.

- Two common methods for assigning a codeword:
  - Fixed-length code (e.g. ASCII 7 bit table https://www.ascii-code.com/ASCII)
  - Variable-length code (Huffman code)

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
32	040	20	00100000	␣	&#32;		Space
33	041	21	00100001	!	&#33;	&excl;	Exclamation mark
34	042	22	00100010	"	&#34;	&quot;	Double quotes (or speech marks)
35	043	23	00100011	#	&#35;	&num;	Number sign
36	044	24	00100100	$	&#36;	&dollar;	Dollar
37	045	25	00100101	%	&#37;	&percnt;	Per cent sign
38	046	26	00100110	&	&#38;	&amp;	Ampersand
39	047	27	00100111	'	&#39;	&apos;	Single quote
40	050	28	00101000	(	&#40;	&lparen;	Open parenthesis (or open bracket)
41	051	29	00101001	)	&#41;	&rparen;	Close parenthesis (or close bracket)

Fixed-length code

- For fixed length code, all codewords have the same number of bits.

- For $n \geq 2$ characters, need $\lceil \lg n \rceil$ bits.

- For example, if we have 7 characters, then we need 3 bits for each character.

$$\lceil \lg n \rceil = \lceil \lg(7) \rceil = \lceil 2.81 \rceil = 3 \text{ bits}$$

$\lceil \quad \rceil$ means round the decimal to upper nearest integer

Fixed-length code

- Suppose that you have a 100,000-character data file that you wish to store compactly.
- There are 6 distinct characters in the file occur with the frequencies (in thousands) given below.
- For six characters, we need 3 bits for each character.
- In total, we need

$$100000 \times 3 = 300000 \quad \text{bits}$$

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

Variable-length code - Huffman code

- Huffman codes are a smart way to compress a data file made up of characters.
- This compression technique can reduce data size significantly, typically saving between 20% to 90% depending on the data's characteristics.

- How Huffman code works?
  - Represent different characters with differing numbers of bits.
  - It looks at how often each character appears in the file—its frequency.
  - Characters that occur frequently get shorter codes, while less common ones get longer ones.

Variable-length code - Huffman code

- How Huffman code works?

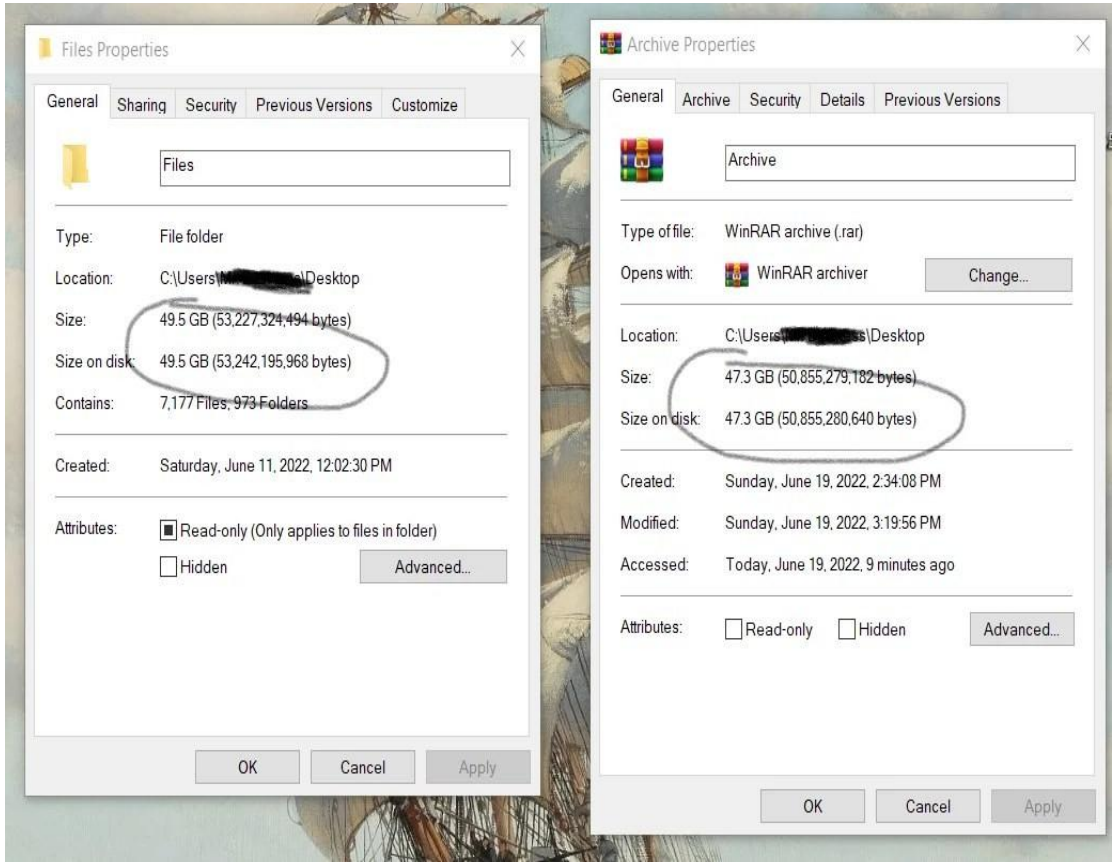
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

  - For this variable length code, we need

$$
\begin{aligned}
45{,}000 \cdot 1 &= 45{,}000 \\
+\,13{,}000 \cdot 3 &= 39{,}000 \\
+\,12{,}000 \cdot 3 &= 36{,}000 \\
+\,16{,}000 \cdot 3 &= 48{,}000 \\
+\;\;9{,}000 \cdot 4 &= 36{,}000 \\
+\;\;5{,}000 \cdot 4 &= 20{,}000 \\
\hline
&= 224{,}000 \text{ bits}
\end{aligned}
$$

Huffman Code – Text Compression

- Huffman coding is used for text compression to reduce the size of text data to save storage space or transmission time.
- Example:
  - Document archiving systems and software use Huffman coding to compress textual content in archived files (e.g., PDF, DOCX) for long-term storage. WinRAR

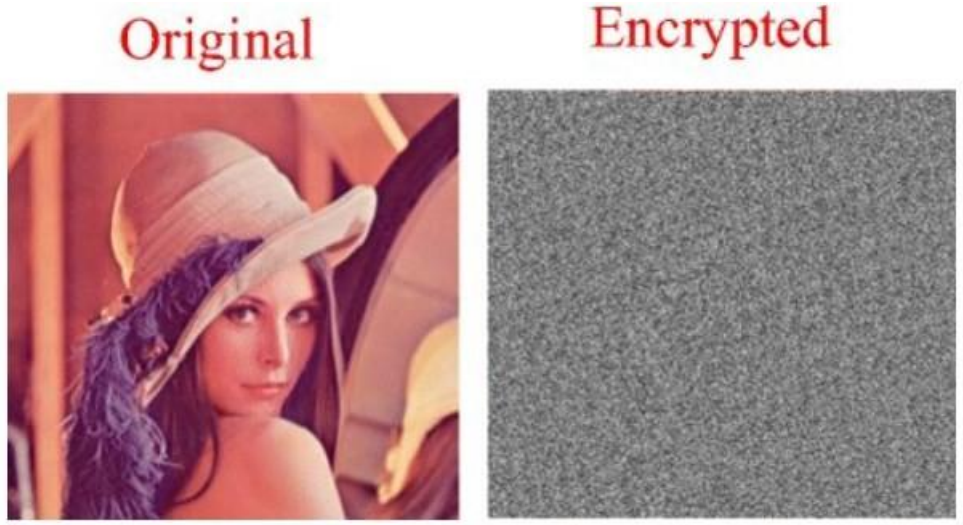


Huffman Code - Image Compression

- Huffman Coding is used in image and video compression techniques like JPEG and MPEG, enabling efficient storage and transmission of multimedia content.
- Example:
  - JPEG Optimizer, Tiny PNG (lossless versions)

Huffman Code - Encryption

- Huffman coding can also be used in encryption algorithms.
- This can make encryption more efficient, especially when dealing with large amounts of data
- it doesn't provide encryption itself; it can be used as part of encryption schemes to compress plaintext before encryption.
- You need the codes to decode (so codes must be non-systematic)



- Example:
  - ADS (Advanced Encryption Standard)
  - RSA (Rivest–Shamir–Adleman), etc.

Huffman Code Procedure

- How to perform Huffman coding?

- Basic steps of Huffman coding are:
  - Frequency Calculation
  - Tree Construction
  - Assigning Code

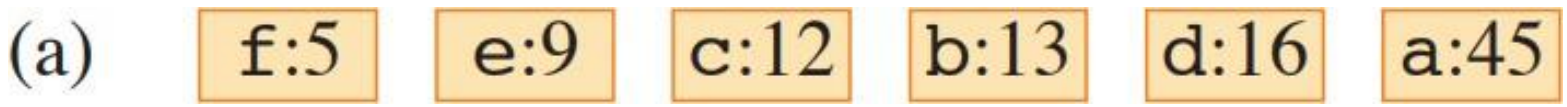
Huffman Code Procedure - Frequency Calculation

- Read the file and calculate the frequency of each character in the file.
- Let's assume that after scanning a data file of 100,000 characters, we find the following character frequencies (in thousands):

Character	Frequency
a	45
b	13
c	12
d	16
e	9
f	5

Huffman Code Procedure – Tree Construction

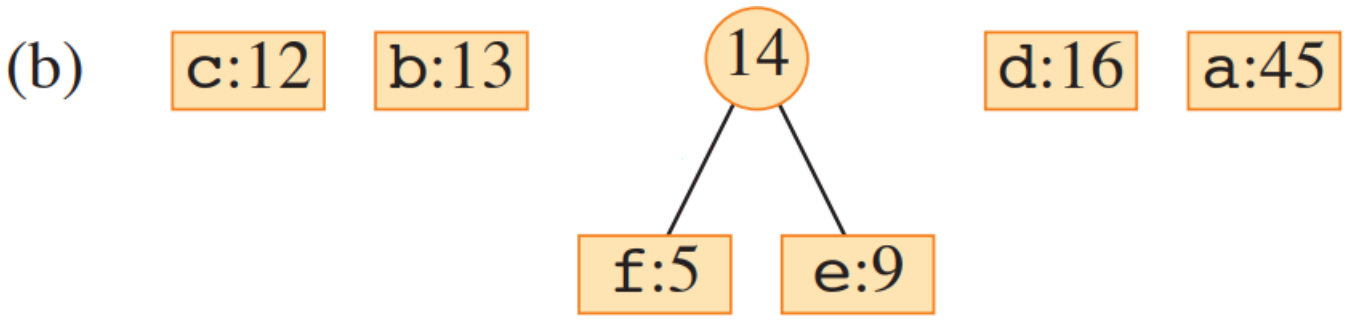
- Create a binary tree for each character that also stores the frequency with which it occurs (as shown below).
- The initial set consists of n = 6 trees, one for each letter.
- Each step merges the two trees with the lowest frequencies.
- Uses min-priority queue (Q), to identify the two least-frequent objects to merge together.
- The contents of the queue sorted into increasing order by frequency.



Huffman Code Procedure – Tree Construction

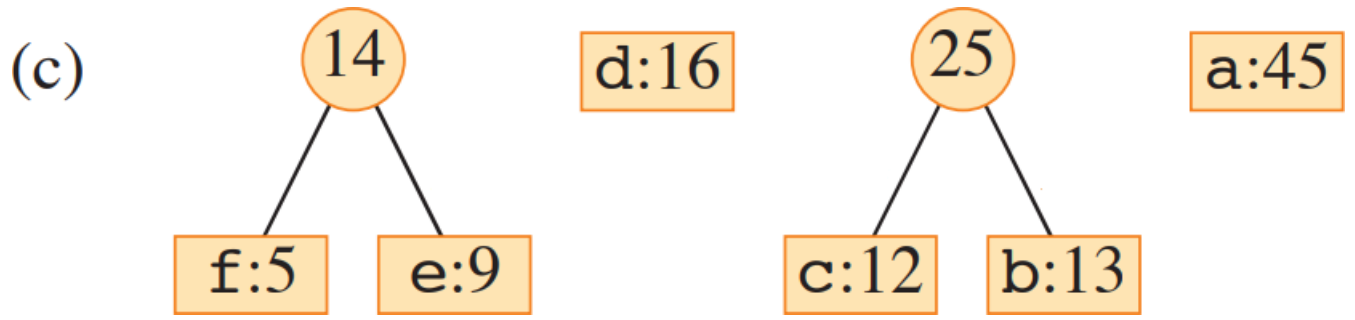
- Selects two trees with the lowest frequency and makes them children of a new tree whose frequency is the sum of the two trees' frequencies.
- Repeat the process, until we left with only one tree.

Iteration 1

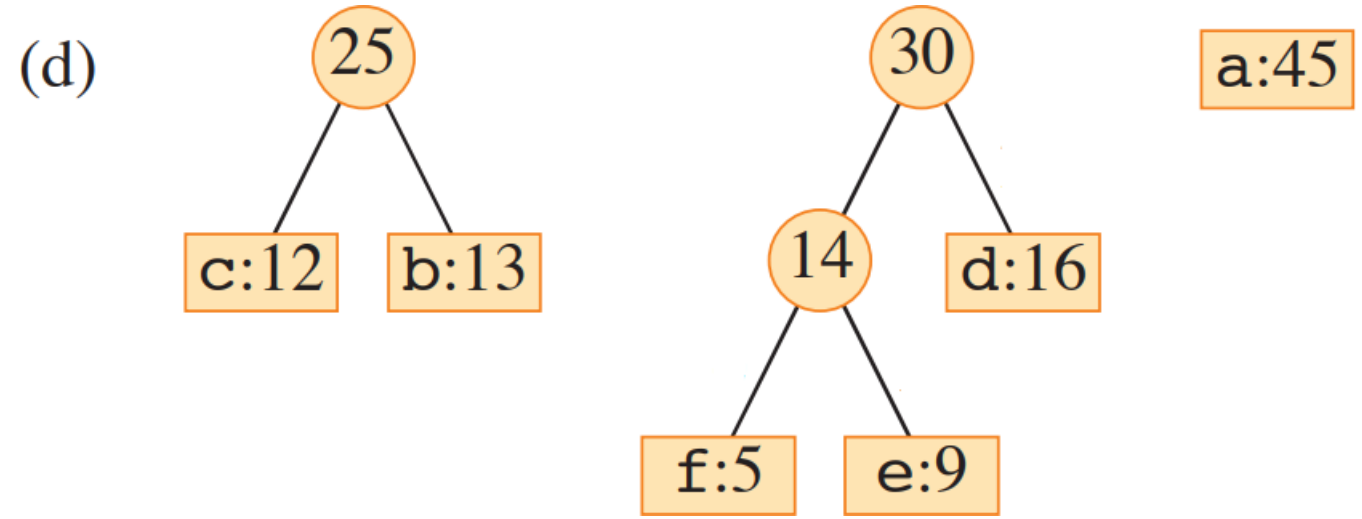


Huffman Code Procedure – Tree Construction

Iteration 2



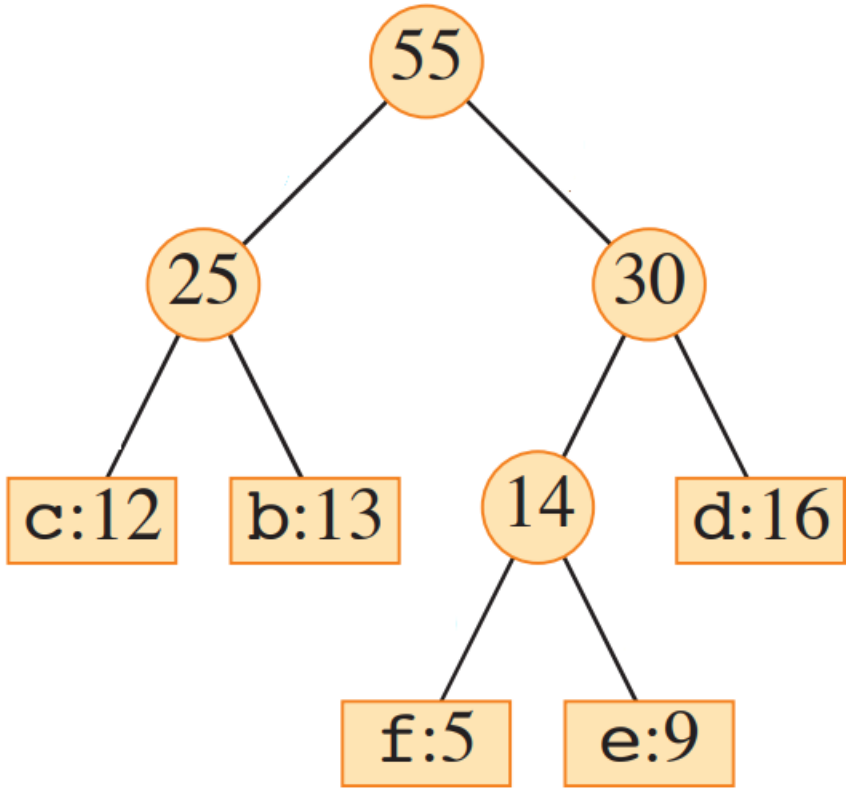
Iteration 3



Huffman Code Procedure – Tree Construction

Iteration 4

(e) a:45



Huffman Code Procedure – Tree Construction

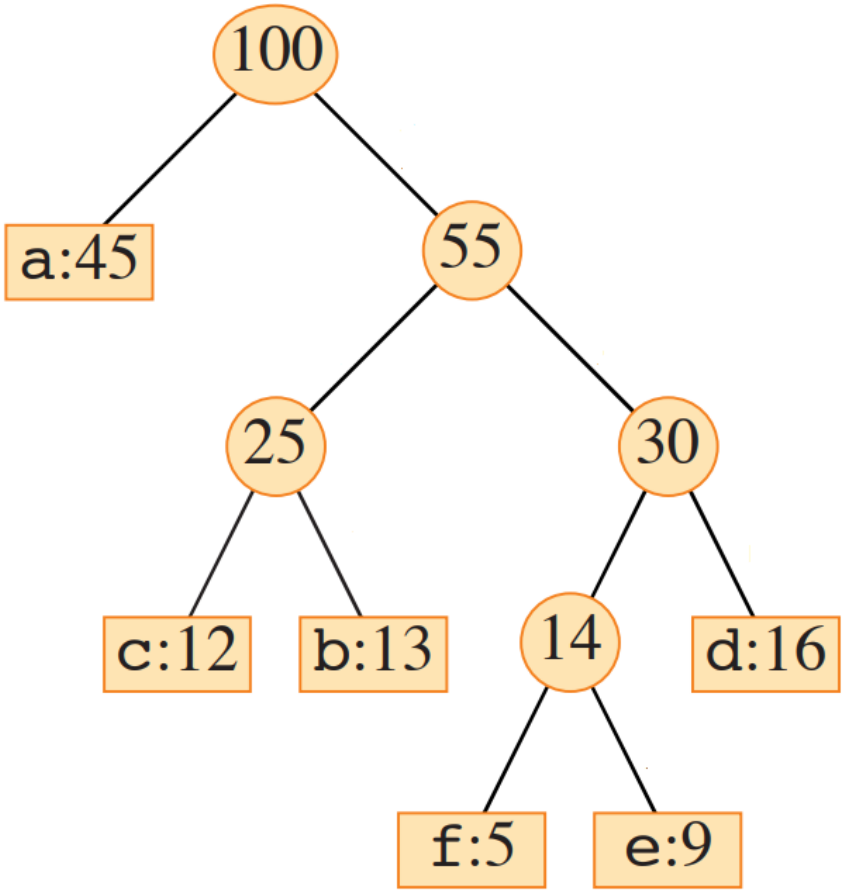
Iteration 5

Here, each leaf has its character and frequency (in thousands). Each internal node (circle) holds the sum of the frequencies of the leaves (rectangles) in its subtree.

If C is the alphabet for the characters, then the tree has $|C|$ leaves and $|C| - 1$ internal nodes.

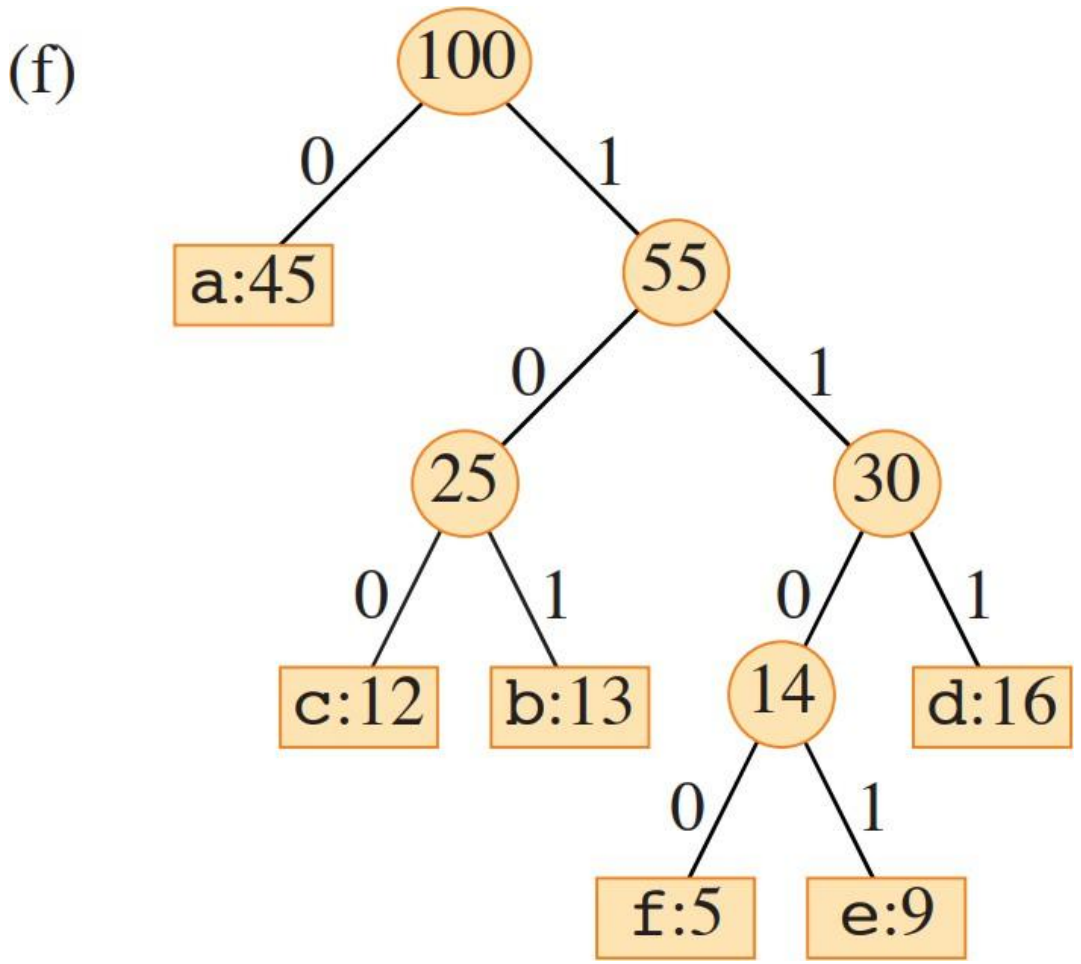
$|C| = 6$ leaves
$|C| - 1 = 6 - 1 = 5$ internal nodes.

(f)



Huffman Code Procedure – Assigning Codes

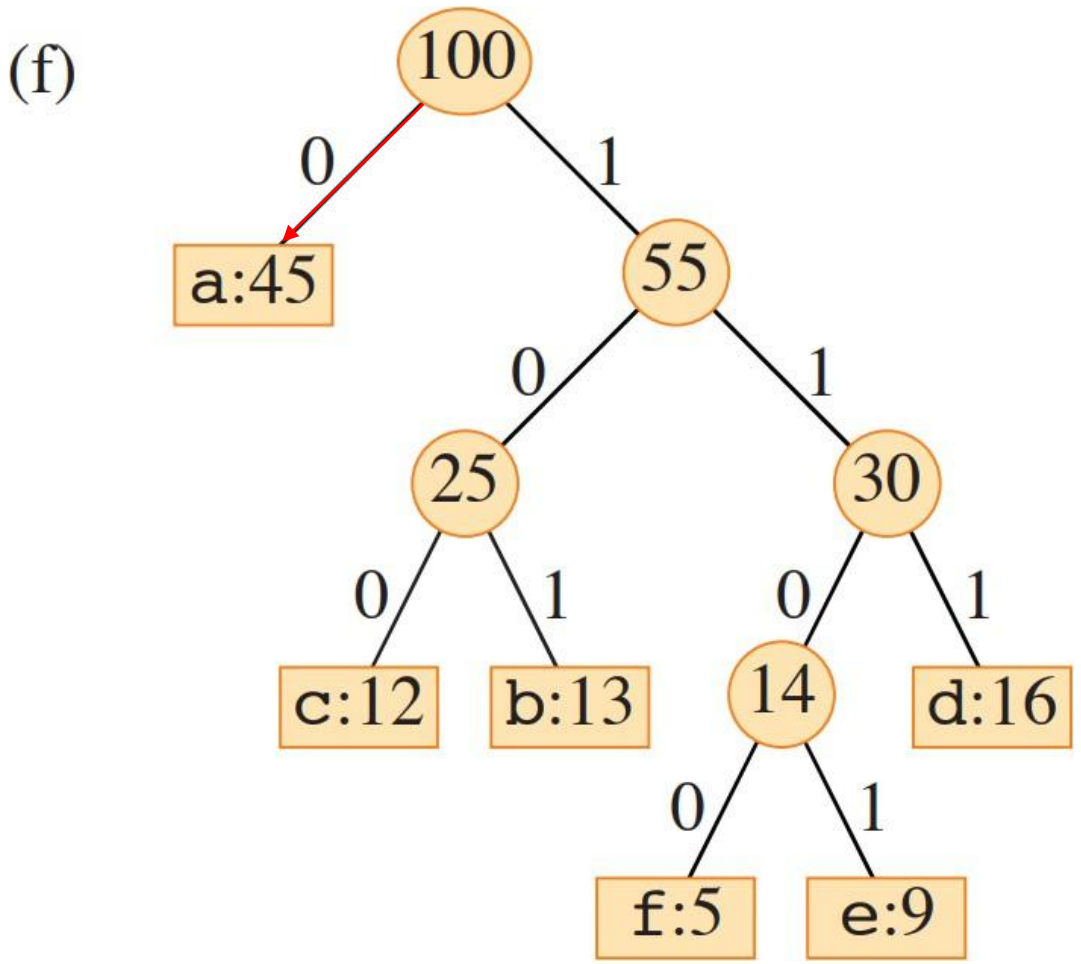
- Once the tree is built, each leaf node corresponds to a letter with a code.
- To determine the code for a particular node, traverse from the root to the leaf node.
- For each move to the left, append a 0 to the code, and for each move to the right, append a 1.



Huffman Code Procedure – Assigning Codes

- To get code of character a, traverse from root node (100) to leaf node (a:45)

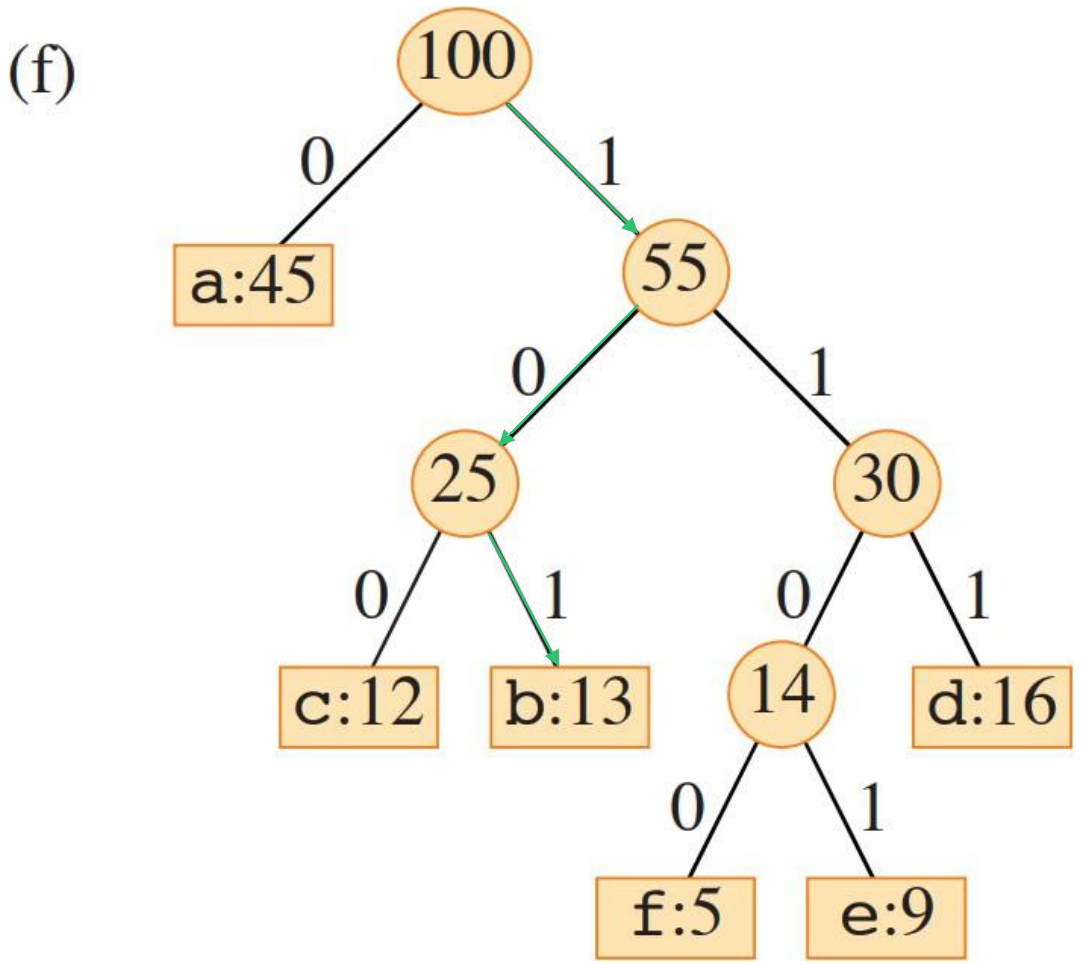
Character	Frequency	Code
a	45	0
b	13	
c	12	
d	16	
e	9	
f	5	



Huffman Code Procedure – Assigning Codes

- To get code of character b, traverse from root node (100) to leaf node (b:13)

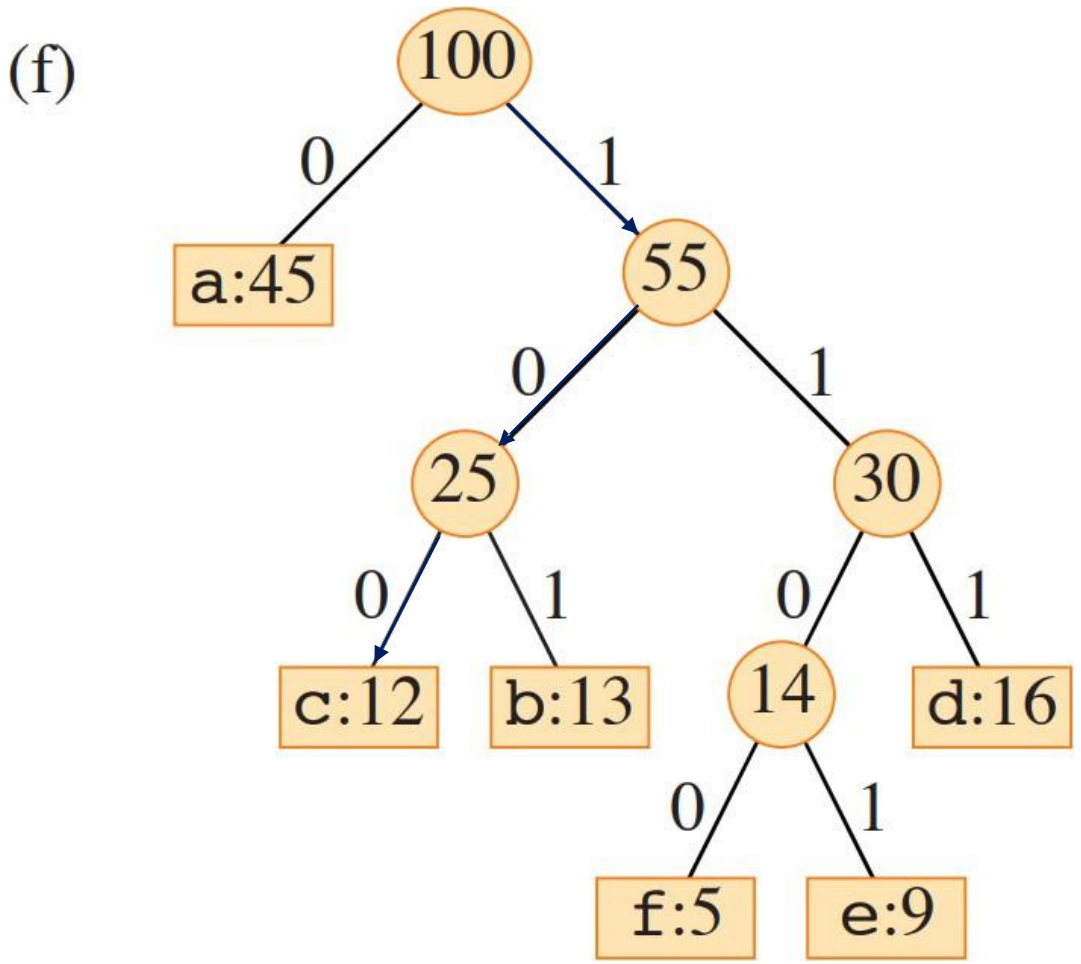
Character	Frequency	Code
a	45	0
b	13	101
c	12	
d	16	
e	9	
f	5	



Huffman Code Procedure – Assigning Codes

- To get code of character c, traverse from root node (100) to leaf node (c:12)

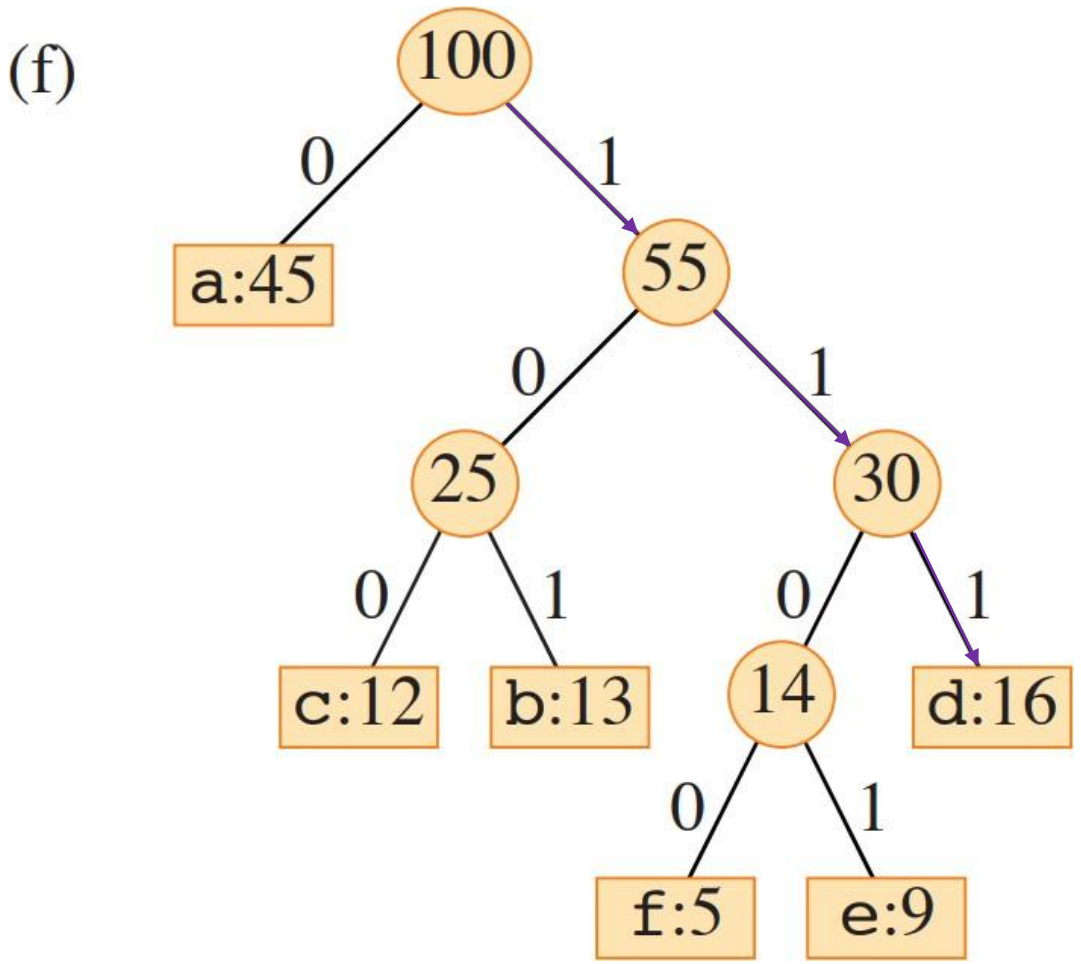
Character	Frequency	Code
a	45	0
b	13	101
c	12	100
d	16	
e	9	
f	5	



Huffman Code Procedure – Assigning Codes

- To get code of character d, traverse from root node (100) to leaf node (d:16)

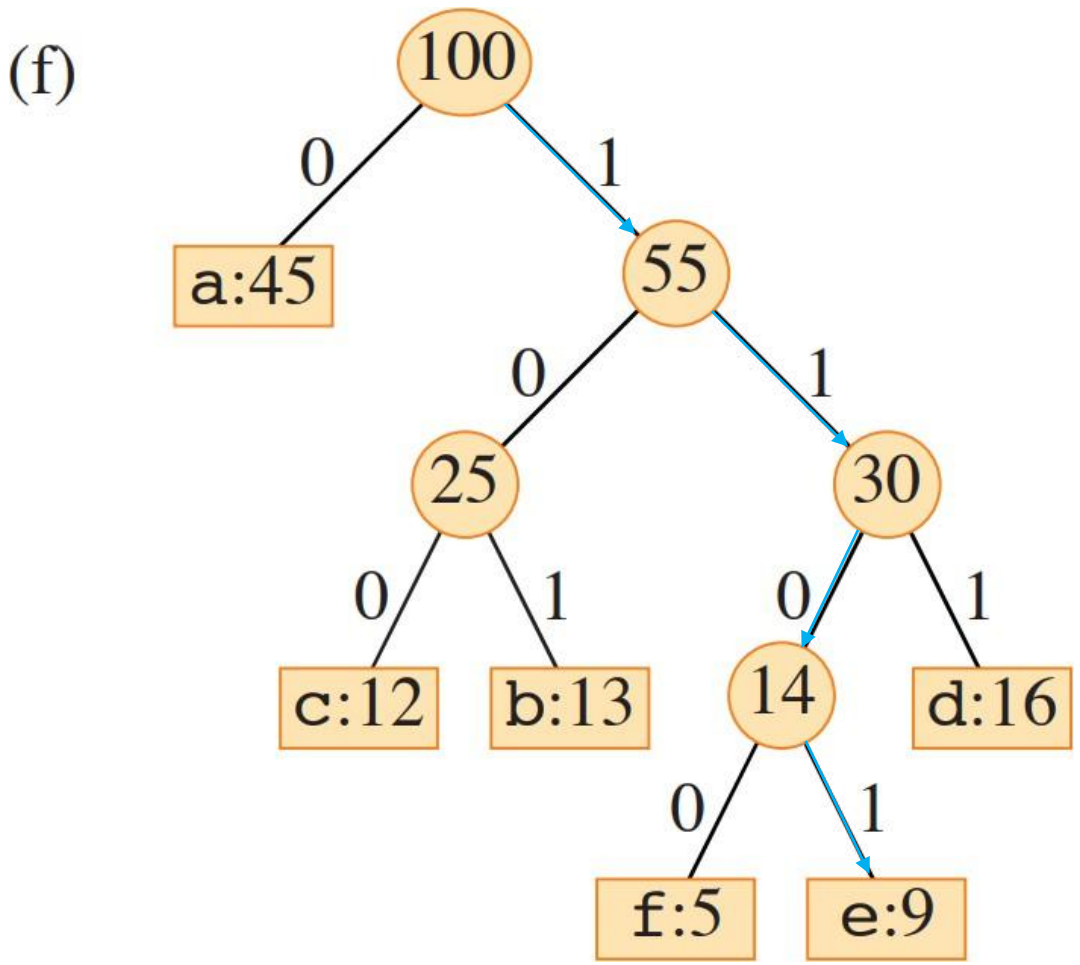
Character	Frequency	Code
a	45	0
b	13	101
c	12	100
d	16	111
e	9	
f	5	



Huffman Code Procedure – Assigning Codes

- To get code of character e, traverse from root node (100) to leaf node (e:9)

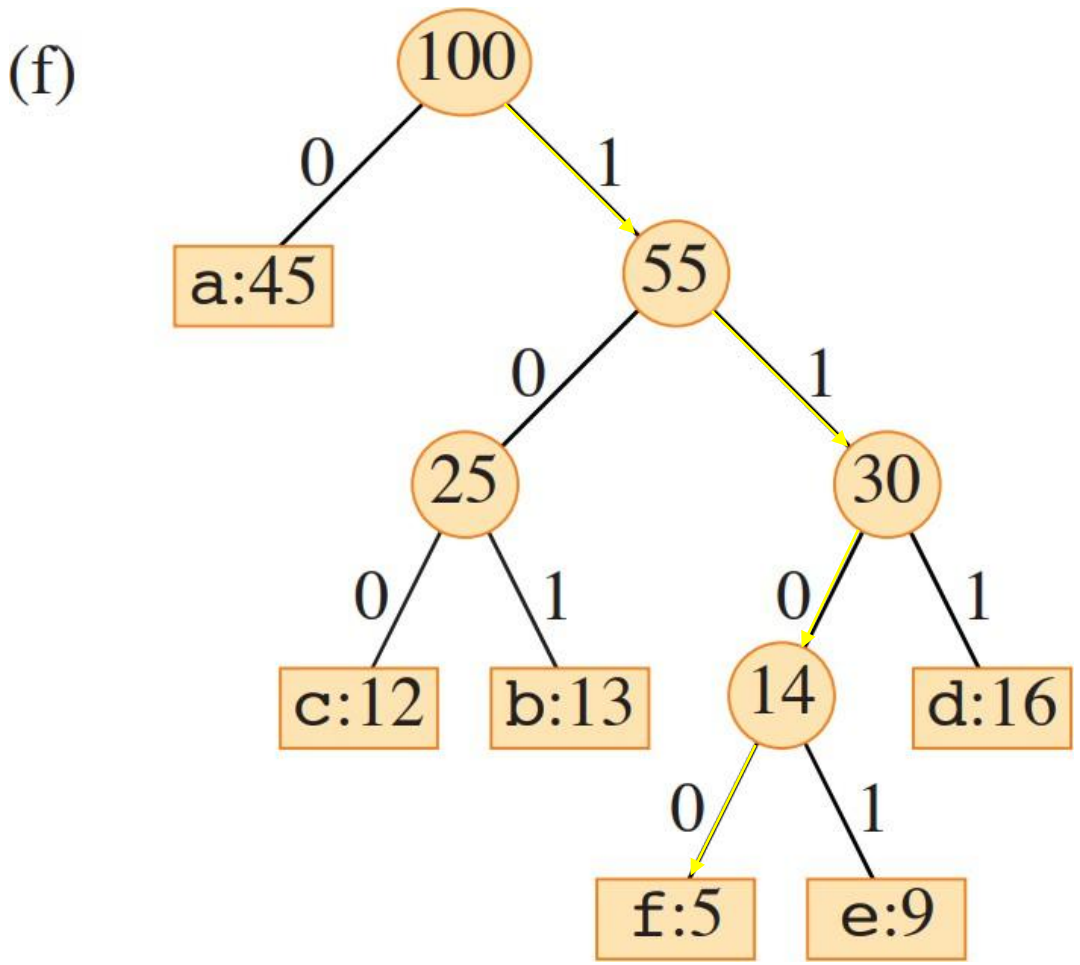
Character	Frequency	Code
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	



Huffman Code Procedure – Assigning Codes

- To get code of character f, traverse from root node (100) to leaf node (e:9)

Character	Frequency	Code
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



Another worked example using a slightly different approach

algs and data struct

Go right to left, increasing height with frequency

Huffman Code Procedure – Pseudocode

```
HUFFMAN(C)
 1  n = |C|
 2  Q = C
 3  for i = 1 to n − 1
 4        allocate a new node z
 5        x = EXTRACT-MIN(Q)
 6        y = EXTRACT-MIN(Q)
 7        z.left = x
 8        z.right = y
 9        z.freq = x.freq + y.freq
10        INSERT(Q, z)
11  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Huffman Code Procedure – Pseudocode

- The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.
- The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner.
- It begins with a set of $|C|$ leaves and performs a sequence of $|C|-1$ "merging" operations to create the final tree.
- The algorithm uses a min-priority queue Q, keyed on the $freq$ attribute, to identify the two least-frequent objects to merge together.
- The result of merging two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
// Create a leaf node for each character and add to min-heap
   FOR each (char, freq) in frequencies:
      node = CREATE_NODE(char, freq)
      INSERT(minHeap, node)

   // Build tree by combining nodes
   WHILE minHeap.size > 1:
      // Get two nodes with lowest frequencies
      left = EXTRACT_MIN(minHeap)
      right = EXTRACT_MIN(minHeap)

      // Create parent node with combined frequency
      parent = CREATE_NODE(null, left.freq + right.freq)
      parent.left = left
      parent.right = right

      // Add parent back to heap
      INSERT(minHeap, parent)

   // Return root of Huffman tree
   RETURN EXTRACT_MIN(minHeap)
```

Huffman Code Procedure - Running time

- Let $n = |C|$, the number of characters in alphabet C.
- O(n) time to initialize min-priority Queue Q (if implemented with a binary min-heap).
- The for loop runs $n - 1$ time, and each INSERT and EXTRACT-MIN call takes O(lg n) time.
- Therefore, the running time of Huffman code is:

$$O(n \lg n)$$

Number of bits - Fixed-length code

- Number of bits to encode a file using Fixed-length code (B_F) are calculated as follows:

$$B_F = n \times b$$

- Where n is the sum of frequencies and b is the number of bits.
- Since there are 100,000 characters in the file n = 100,000.
- Let's assume each character is stored with a fixed three-bit code b = 3.
- Multiply total frequencies by 3, the total number of bits used are

$$B_F = 100000 \times 3$$
$$B_F = 300{,}000$$

Number of bits - Huffman Code

- Number of bits to encode a file using Huffman coding (B_T) are calculated as follows:

$$B_T = \sum_{c \in C} \mathrm{f}_c \times d_T(c)$$

- where f_c is the frequency of each character $c \in C$
- dT_c is the depth of each character c in tree T (which equals the length of c's codeword).

Worked example

Fuzhou hills bloom daily

$$\sum^{n}$$

1. Table of characters with counts
2. Start with those of the least frequency at bottom right
3. Pair them up, from right to left, putting sum in parent node
4. Once all leafs paired, pair parents
5. Once completed note 0 of every left branch and 1 on every right branch
6. Pathway to every character is their code

Number of bits - Huffman Code

$$B_T = \sum_{c \in C} f_c \times d_T(c)$$

$$B_T = 244000$$

$$\begin{aligned} 45{,}000 \cdot 1 &= 45{,}000 \\ +\ 13{,}000 \cdot 3 &= 39{,}000 \\ +\ 12{,}000 \cdot 3 &= 36{,}000 \\ +\ 16{,}000 \cdot 3 &= 48{,}000 \\ +\ 9{,}000 \cdot 4 &= 36{,}000 \\ +\ 5{,}000 \cdot 4 &= 20{,}000 \\ \hline &= 224{,}000 \text{ bits} \end{aligned}$$

Character	Frequency (fc)	Code	length (dT_c)
a	45	0	1
b	13	101	3
c	12	100	3
d	16	111	3
e	9	1101	4
f	5	1100	4

Calculating Bits Saved

- To calculate the number of bits saved, we subtract the number of bits using Huffman-code (B_T) from the number of bits using fixed-length code (B_F).

$$\text{Bits saved} = B_F - B_T$$

- $B_F = 300000$ and B_T = 244000.

$$\text{Bits saved} = 300000 - 244000$$

$$\text{Bits saved} = 56000$$

- Therefore, using Huffman coding, we saved 56000 bits.

Compression Ratio

- The compression ratio is a measure used to evaluate the effectiveness of a data compression algorithm.
- It quantifies the reduction in size achieved by compressing data.
- It is calculated as the ratio between the original size of the data and the compressed size.

$$CR = \frac{\text{Original Data Size}}{\text{Compressed Data Size}}$$

Compression Ratio

- **CR > 1**: A **higher ratio** indicates more effective compression, meaning the compressed file is significantly smaller than the original file.

- **CR = 1**: A ratio of 1 indicates no compression at all (the compressed size is equal to the original size).

- **CR < 1**: A ratio less than 1 (although this is often expressed instead as a negative compression rate) indicates that the "compression" process actually made the data larger, which can happen with very small files or files that are already in a highly compressed format.

Compression Ratio

- In our example, the original size of a file is 300,000 and the compressed size is 2440000, the compression ratio would be:

$$CR = \frac{B_F}{B_T}$$

$$CR = \frac{300{,}000}{244{,}000}$$

$$CR = 1.23$$

- A CR > 1 indicates effective compression, meaning the compressed file using Huffman code is significantly smaller than the original file using fixed-length code.

Huffman Code - Exercise

- Draw a Huffman code tree based on the following frequency of characters.

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

- Assign codes to each character.
- Calculate the resulting Huffman compression ratio, assuming an original fixed-length coding of 8 bits per character.

Problem

Huffman coding less useful if evenly distributed character counts

Alternatives

- Arithmetic Coding
- Lossy encoding (jpeg)

Contemporary Data Structures and Algorithms

Transformer-Based Algorithms

Diffusion Models

Quantum Algorithms

Federated Learning Algorithms

Graph Neural Network Algorithms