Data Structures and Algorithms

Chapter 13

Greedy Algorithms

- Consider the algorithms of the class project.
- The problem is: Given a start node *S*, find the a shortest path from *S* to a destination node *D*.
- We can solve this problem by
  - Find all possible paths from *S* to *D*.
  - Select a path with the shortest length.
- This approach guarantees that we find a solution, but it could be expensive.
- A greedy approach: Beginning at *S*, select the next node which is best at that moment, such as based on *direct distances*.
- Another simple example: *coin changing* problem

Greedy Algorithms

- When we solve an optimization problem, we need to make a series of choices.
- When making a choice, the greedy method considers all options that are “available at that moment” and chooses the best option among them.
- In other words, it chooses a “locally optimal” option.
- The greedy method does not always lead to a global optimal solution.
- However, for many practical problems, the greedy method gives us a global optimal solution.

- Will describe the *Huffman code* algorithm, which is a greedy algorithm.

Huffman Code - Introduction

- A data is considered as a sequence of characters.
- Each character is encoded to a unique binary string, called a *codeword*.
- Example:
  - ‘A’ is encoded to a codeword 0000
  - ‘B’ is encoded to a codeword 0001
  - and so on
- Decoding: Converting a codeword to the initial character.

Huffman Code - Introduction

- There are different ways of encoding characters to binary strings.
- A fixed-length code uses the same number of bits for different characters.
- Example of a fixed-length code: ASCII code.
- A variable-length code uses different number of bits for different characters.

Huffman Code - Introduction

- Fixed-length code vs. variable-length code
  - Fixed-length code: Uses the same number of bits for all characters.
  - Variable-length code: Uses different number of bits for different characters.
- Prefix code: No codeword is a prefix of some other codeword.
- For example, if the codeword for ‘X’ is 10100 and the codeword for ‘Y’ is ‘101”, then this code is NOT a prefix code (because 101 is a prefix of 10100)
- Prefix codes simplify the decoding process.

Huffman Code - Introduction

- A goal of data compression: Minimize the size of the compressed data (where each character is represented by a codeword).
- The Huffman code is a *variable-length*, *prefix* code used for data compression.
- It uses a smaller number of bits for a character that appears in the document with a high frequency and uses a larger number of bits for a character that appears rarely.

Huffman Code - Introduction

- The following table shows the frequency of occurrences of each character in a given data and two coding schemes.

	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

Huffman Code - Introduction

- The fixed-length code requires 300,000 bits (3 bits X 100,000 characters).

- The variable-length code requires less number of bits:

  $45000 \cdot 1 + 13000 \cdot 3 + 12000 \cdot 3 + 16000 \cdot 3 + 9000 \cdot 4 + 5000 \cdot 4 = 224{,}000$ bits

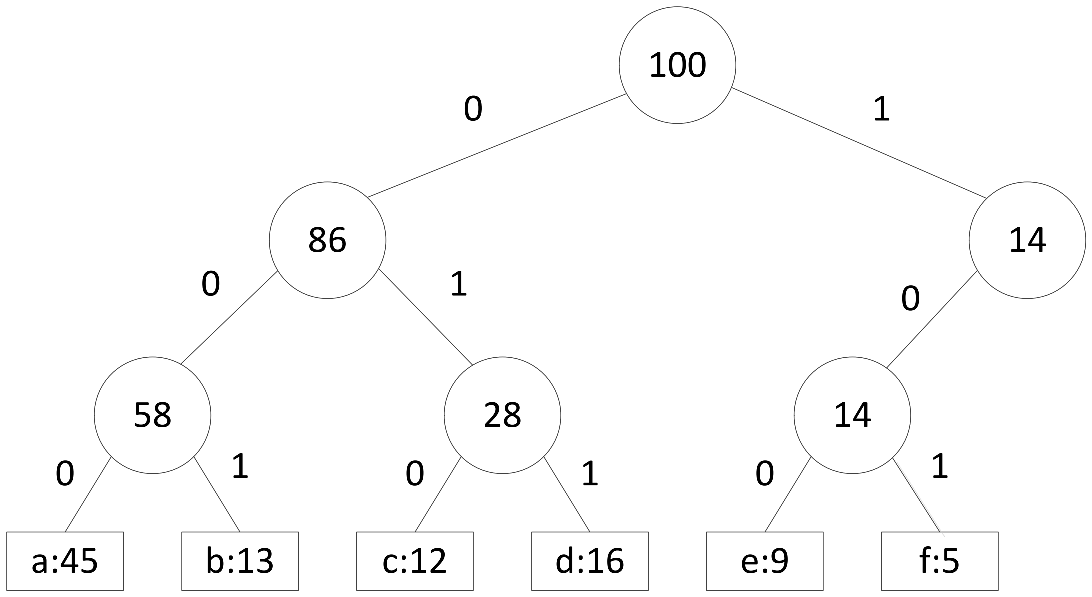
  This code happens to be an optimal code for the given data.

Huffman Code - Introduction

- Huffman code algorithm is a greedy algorithm that constructs an *optimal prefix code* called *Huffman code*.
- Encoding: Represent each character in the data with the corresponding codeword.
- Decoding: Convert an encoded data to the original data. This can be done efficiently using a binary tree.

Huffman Code - Decoding

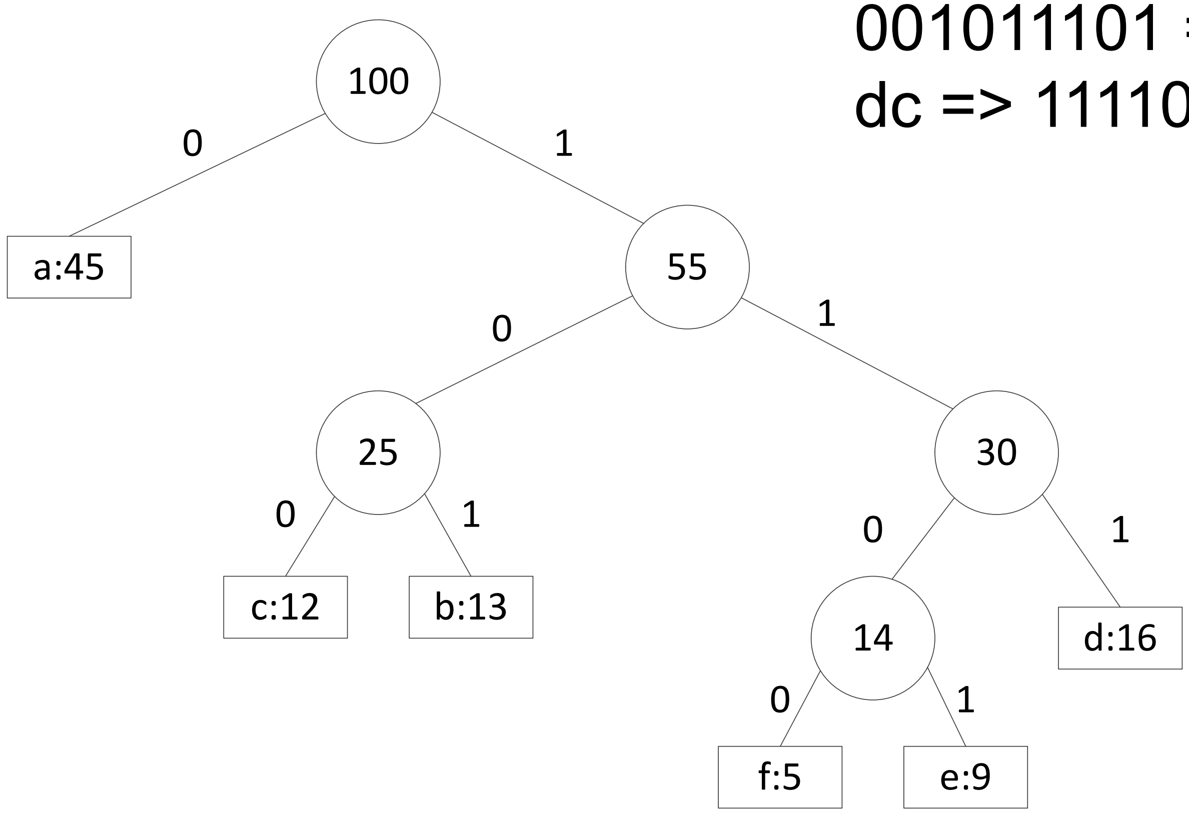
- Coding tree for the fixed-length code (of the above example)



Huffman Code - Decoding

- Coding tree for the variable-length code, Huffman code (of the above example)

001011101 => aabe
dc => 111100



Huffman Code - Decoding

- In a binary tree for an optimal code, each node has exactly two children.
- Decoding:
  - Begin at the root and scan the binary code.
  - If a bit is 0, go down to the left. If a bit is 1, go down to the right.
  - When you are at a leaf node, the decoding of one character is done and the character is shown in the leaf node.
  - Go back to the root and repeat the same with the remaining bit string.

Huffman Code - Decoding

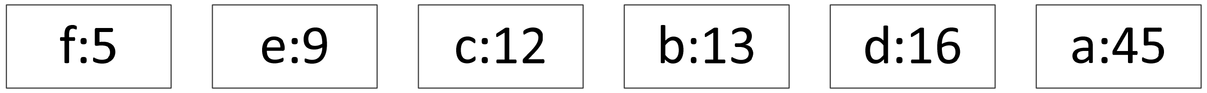
- Decoding of 001011101 (Huffman code):
  - Scanning the first bit, 0, takes you to a leaf node with the character *a*. So, it is decoded as *a*.
  - Next 0 is also decoded as *a*.
  - The next three bits 101 leads to *b*.
  - The next four bits 1101 decodes to *e*.
  - So, the decoded string is *aabe*.

Huffman Code - Encoding

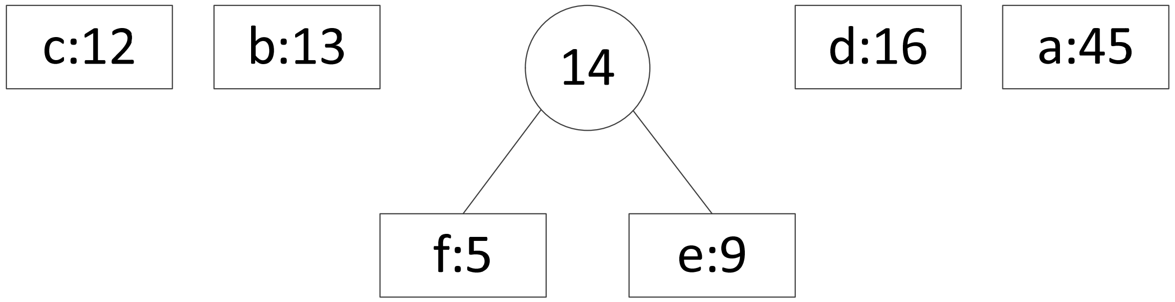
- To encode a character, follow the path from the root to the leaf corresponding to the character, and concatenate the bits along the path.
- Example: encoding *dc*
  - The path from the root to the leaf with *d*: 111
  - The path from the root to the leaf with *c*: 100
  - So, the *dc* is encoded to 111100

Constructing a Huffman Code

- Illustration
  - (a) Initial Q (which is a priority queue)

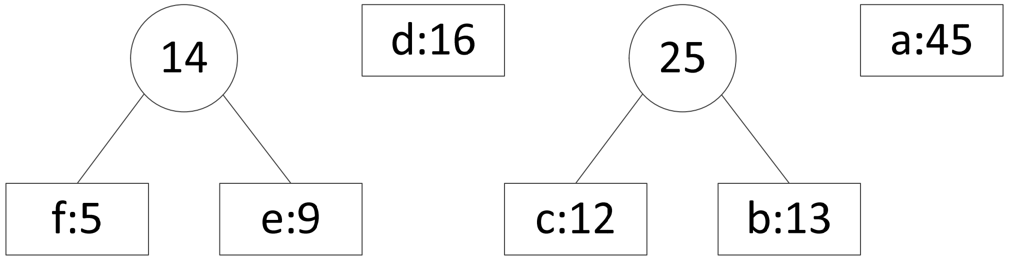


  - (b) (f:5) and (e:9) are extracted, merged, and inserted into Q.

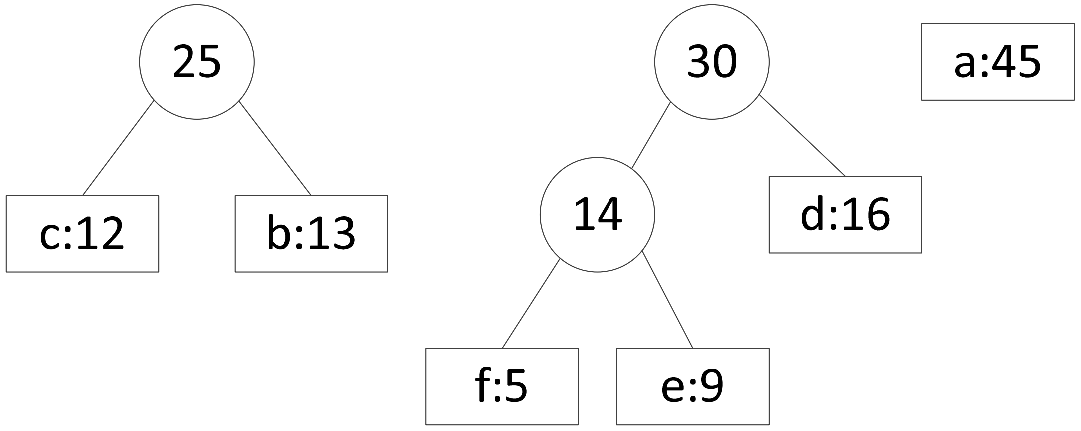


Constructing a Huffman Code

(c) (c:12) and (b:13) are extracted, merged, and inserted into Q.

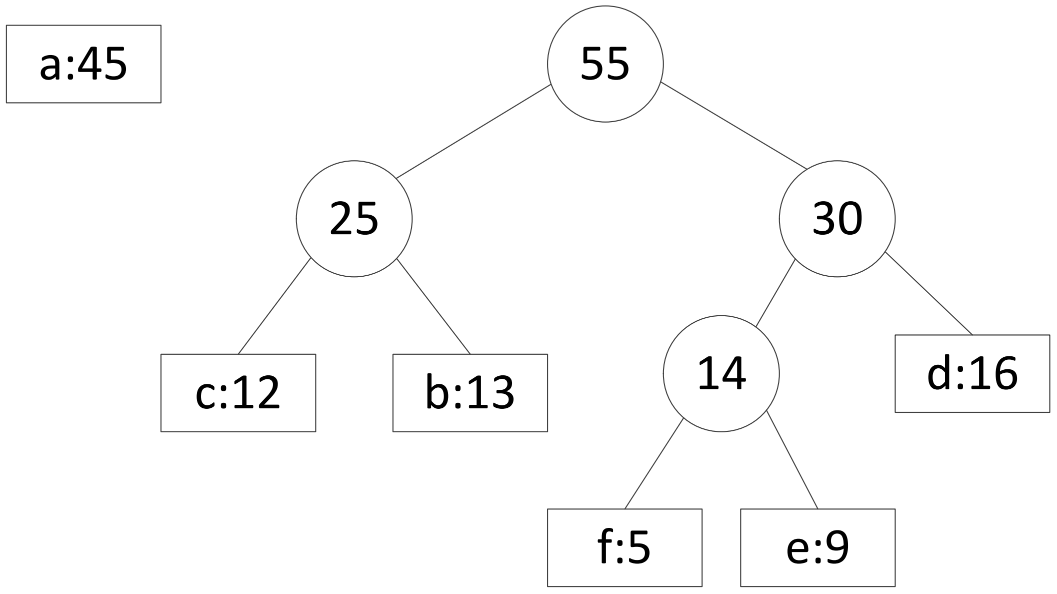


(d) ((f:15, e:9):14) and (d:16) are extracted, merged, and inserted into Q.



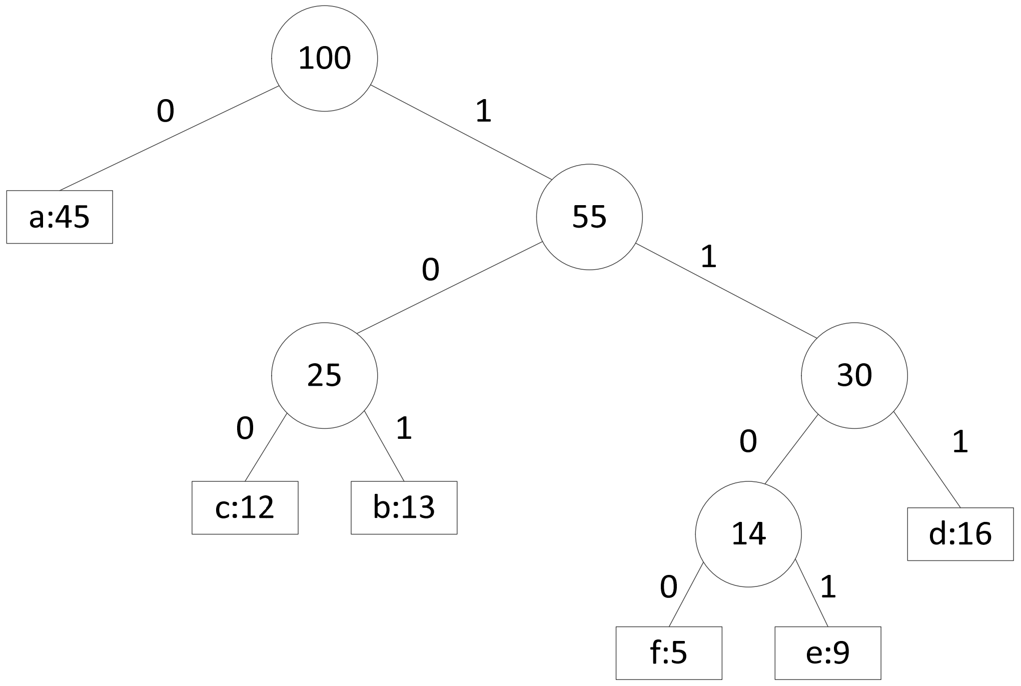
Constructing a Huffman Code

(e) ((c:12, b:13):25) and (((f;15, e:9):14, d:16):30) are extracted, merged, and inserted into Q.



Constructing a Huffman Code

(f) (a:45) and ((c:12, b:13):25, (((f:5, e:9):14, d:16):30):55) are extracted, merged, and inserted into Q.



Dynamic Programming

- Refers to a technique or an approach, not an algorithm.
- Solves problems by combining solutions to subproblems (like divide-and-conquer).
- If subproblems are not independent, some subproblems are solved multiple times.
- Dynamic programming approach:
  - Bottom-up approach: Problems are solved in the increasing order of size (i.e., smallest problem first, followed by the next smallest problem, and so on).
  - Each subproblem is solved once and the solution is stored in a table.
- Typically used for optimization problems.

Dynamic Programming –World Series

- Consider the following problem (from Aho, Hopcroft, and Ullman):
  - Two baseball teams *X* and *Y* are competing for the World Series championship.
  - A team wins the championship title if it wins four out of seven games.

  - *P*(*i*, *j*) is defined as: the probability that one of the teams, say *X*, will eventually win the championship title, given that *X* still needs to win *i* more games to win the title and *Y* still needs to win *j* more games to win the title.

Dynamic Programming

- Consider the following problem (continued):
  - Example: *X* won 1 game and *Y* won 2 games. Then, *X* needs 3 more games and Y needs 2 more games, and the probability that *X* will win the championship title is denoted $P(3, 2)$.
  - We assume that two teams are equally likely to win any particular game.

  - Two extreme cases

    $P(0, j) = 1$ for any $j > 0$ // *X* won the championship

    $P(i, 0) = 0$ for any $i > 0$ // *Y* won the championship

Dynamic Programming

- Consider the following problem (continued):
  - In general, we can calculate $P(i, j)$ recursively as follows:

$$
\begin{aligned}
P(i, j) &= 1, \text{ if } i = 0 \text{ and } j > 0 \\
&= 0, \text{ if } i > 0 \text{ and } j = 0 \\
&= (P(i-1, j) + P(i, j-1)) / 2, \text{ if } i > 0 \text{ and } j > 0
\end{aligned}
$$

  - This is a divide-and-conquer approach.
  - But, some subproblems are solved multiple times.

Dynamic Programming

- Consider the following problem (continued):
  - For example,

    $P(7, 7) = (P(6, 7) + P(7, 6)) / 2$

    $P(6, 7) = (P(5, 7) + P(6, 6)) / 2$

    $P(7, 6) = (P(6, 6) + P(7, 5)) / 2$

  - In this example, $P(6, 6)$ is calculated more than once.

Dynamic Programming

- Dynamic programming approach:
  - We solve smaller problems first (smaller problems refer to $P(i, j)$ with small i and j).
  - Store the results in a table.
  - When we solve a larger problem, we use the solutions to smaller problems, which are stored in the table.

Dynamic Programming

- Illustration
  - First, we solve *P*(0, *j*) for all *j* (i.e., *j* = 1, 2, 3, 4, 5, 6) and solve *P*(*i*, 0) for all *i* (i.e., *i* = 1, 2, 3, 4, 5, 6) and store them in a table:

P(*i, j*)

						1	6
						1	5
						1	4
						1	3
						1	2
						1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

$\uparrow j$

$\leftarrow i$

Dynamic Programming

- Illustration (continued)
  - Next,
    - $P(1, 1) = (P(0, 1) + P(1, 0)) / 2 = (1 + 0) / 2 = 1/2;$
    - $P(1, 2) = (P(0, 2) + P(1, 1)) / 2 = (1 + 1/2) / 2 = 3/4;$
    - $P(2, 1) = (P(1, 1) + P(2, 0)) / 2 = (1/2 + 0) / 2 = 1/4;$

$P(i, j)$

						1	6	
						1	5	
						1	4	$\uparrow$
						1	3	
					3/4	1	2	
				1/4	1/2	1	1	j
0	0	0	0	0	0		0	
6	5	4	3	2	1	0		

$\leftarrow \; i$

Dynamic Programming

- Illustration (continued)
  - Next,
    - $P(1, 3) = (P(0, 3) + P(1, 2)) / 2 = (1 + 3/4) / 2 = 7/8;$
    - $P(2, 2) = (P(1, 2) + P(2, 1)) / 2 = (3/4 + 1/4) / 2 = 1/2;$
    - $P(3, 1) = (P(2, 1) + P(3, 0)) / 2 = (1/4 + 0) / 2 = 1/8;$

$P(i, j)$

						1	6
						1	5
						1	4
					7/8	1	3
				1/2	3/4	1	2
			1/8	1/4	1/2	1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

$\uparrow j$

$\leftarrow i$

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.
- A.V. Aho, J.E. Hopcroft, and J.D. Ullman, “Data Structures and Algorithms,” Addison-Wesley, 1983, pp. 312 – 314.