COMP90038 Algorithms and Complexity

Lecture 21: Huffman Encoding for Data Compression

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

- The SES is open now. Please take a time to review this subject. Constructive feedback is greatly appreciated

- The final exam has been scheduled for 1:15PM on Tuesday 12th of November at the Royal Exhibition Building
  - There are more instructions available in the LMS, including sample exams

- Assignment 2 is due next Sunday at 11:59PM
  - We will provide sample answers through the LMS once the results are released (likely by the week of the 4th of November)

- Next week we will use a lecture for a quick review of the content
  - Only examinable topics will be discussed in the review
  - Please bring questions ready
  - We will announce the methodology soon

Data compression

- From an information-theoretic point of view, most computer files contain much redundancy

- Compression is used to store files in less space
  - For text files, savings up to 50 are common
  - For binary files, savings up to 90 are common

- Savings in space mean savings in time for file transmission

Run-length encoding

- For a text with long runs of **repeated characters**, we could compress by counting the runs. For example:

AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

- can then be encoded as:

4A3BAA5B8CDABCB3A4B3CD

- This is not useful for normal text. However, for **binary files** it can be very effective.

Run-length encoding

```
000000000000000000000000000011111111111111000000000
000000000000000000000000001111111111111111110000000
000000000000000000000001111111111111111111111110000
000000000000000000000011111111111111111111111111000
000000000000000000001111111111111111111111111111110
000000000000000000011111110000000000000000001111111
000000000000000000011111000000000000000000000011111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000001111000000000000000000000001110
000000000000000000000011100000000000000000000111000
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011000000000000000000000000000000000000000000000011
```

Variable-length encoding

- Fixed-length encoding uses a static number of symbols (bits) to represent a character.
  - For example, the ASCII code uses 8 bits per character.

- Variable-length encoding assigns shorter codes to common characters
  - In English, the most common character is **E**, hence, we could assign **0** to it
  - However, no other character code can start with **0**

No character's code should be a prefix of some other character's code

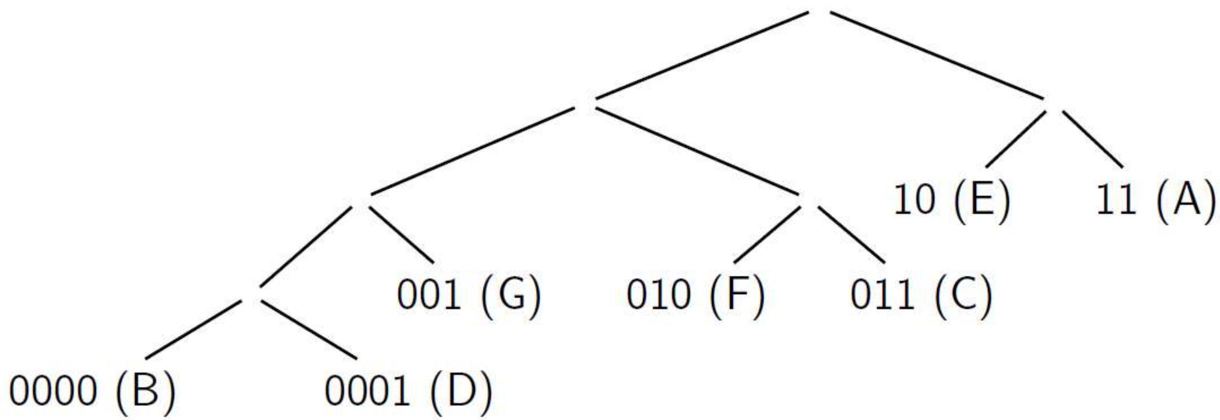
Variable-Length Encoding

- Suppose our alphabet is {A,B,C,D,E,F,G}
- We analyzed a text and found the following number of occurrences
- The last column shows some sensible codes that we may use for each symbol
  - Symbols with higher occurrence have shorter codes

SYMBOL	OCCURRENCE	CODE
A	28	11
B	4	0000
C	14	011
D	5	0001
E	27	10
F	12	010
G	10	001

Tries for Variable-Length Encoding

- A **trie** is a binary tree used on search applications
- To search for a key we look at individual **bits** of a key and descend to the **left** whenever a bit is **zero** and to the right whenever it is **one**
- Using a trie to determine codes means that no code will be the prefix of another



Encoding messages

- To encode a message, we just need to concatenate the codes. For example:

F	A	C	E
010	11	011	10

B	A	G	G	E	D
0000	11	001	001	10	0001

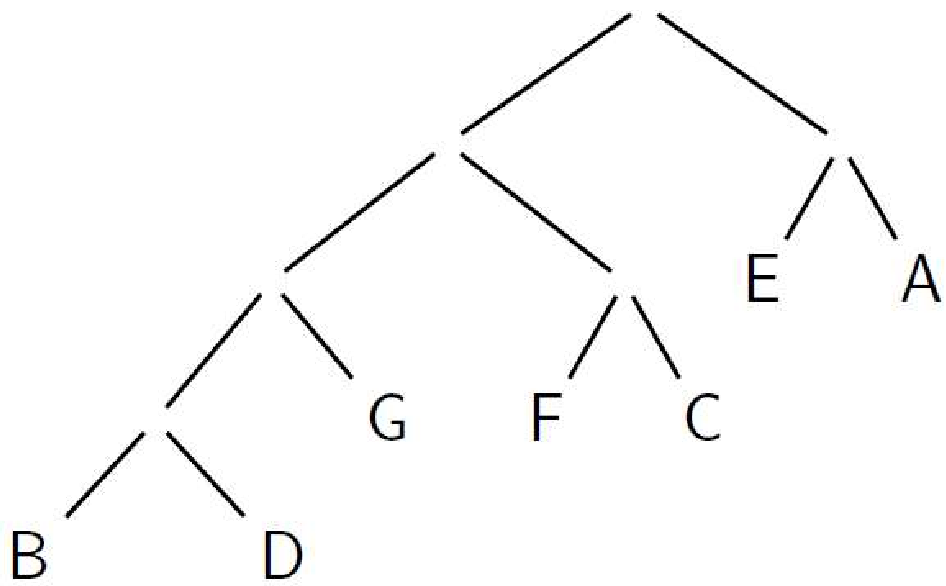
- If we were to assign three bits per character, FACE would use 12 bits instead of 10. For BAGGED there is no space savings

SYMBOL	CODE
A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

Decoding messages

- Try to decode **00011001111010** and **000011000100110** using the trie
  - Starting from the root, print each symbol found as a leaf
  - Repeat until the string is completed

- Remember the rules: Left branch is 0, right branch is 1



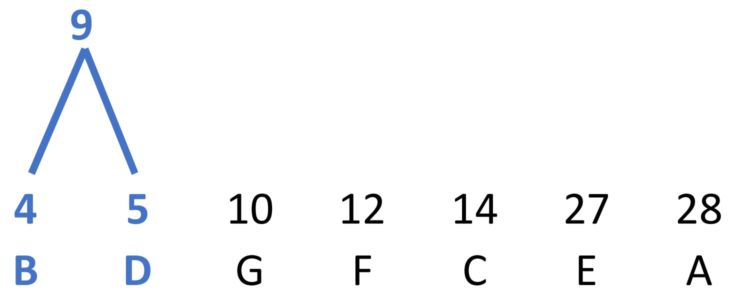
Huffman encoding: choosing the codes

- Sometimes (for example for common English text) we may know the frequencies of letters fairly well
- If we don't know about frequencies then we can still count all characters in the given text as a first step
- But how do we assign codes to the characters once we know their frequencies?
  - By repeatedly selecting the two smallest weights and fusing them
- This is **Huffman's algorithm** – another example of a **greedy method**
  - The resulting tree is a **Huffman tree**

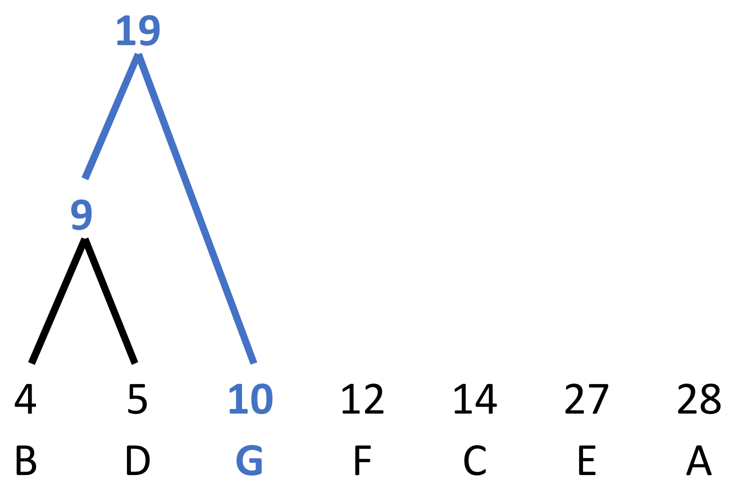
Huffman Trees (example)

4	5	10	12	14	27	28
B	D	G	F	C	E	A

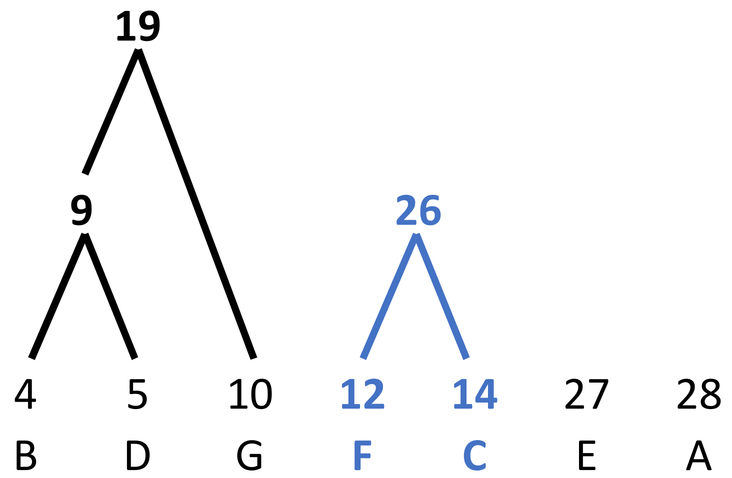
Huffman Trees (example)



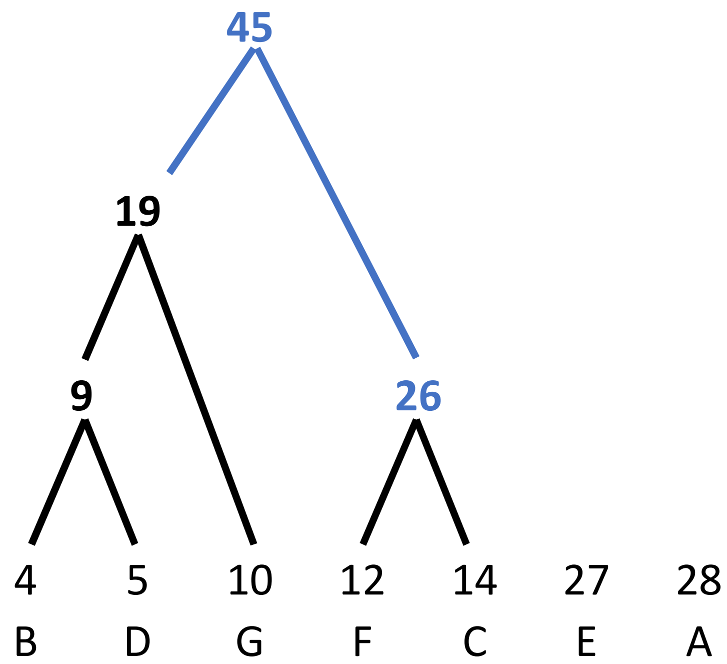
Huffman Trees (example)



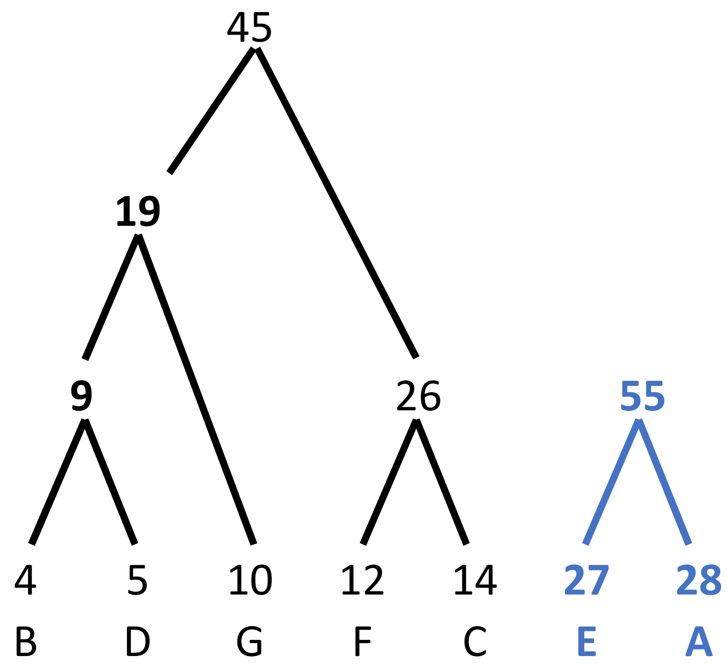
Huffman Trees (example)



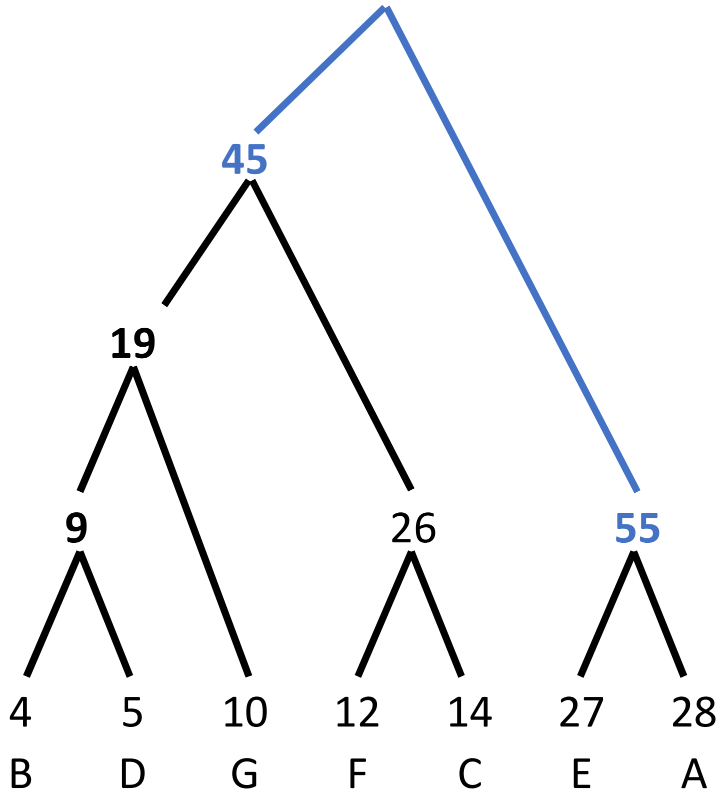
Huffman Trees (example)



Huffman Trees (example)



Huffman Trees (example)



Compressed Transmission

- If the compressed file is being sent from one party to another, the parties must agree about the codes used
  - For example, the trie can be sent along with the message

- For long files this extra cost is negligible

- Modern variants of Huffman encoding, like **Lempel-Ziv compression**, assign codes not to individual symbols but to sequences of symbols

An exercise

- Construct the Huffman code for data in the table, placing in the tree from left to right [A,B,D,_,C]
- Then, encode **ABACABAD** and decode **100010111001010**

SYMBOL	FREQUENCY	CODE
A	0.40	
B	0.10	
C	0.20	
D	0.15	
_	0.15	

Next lecture

- NP-completeness