

PG4200: Algorithms And Data Structures

Lesson 12: Data Compression

Dr. Andrea Arcuri

Data

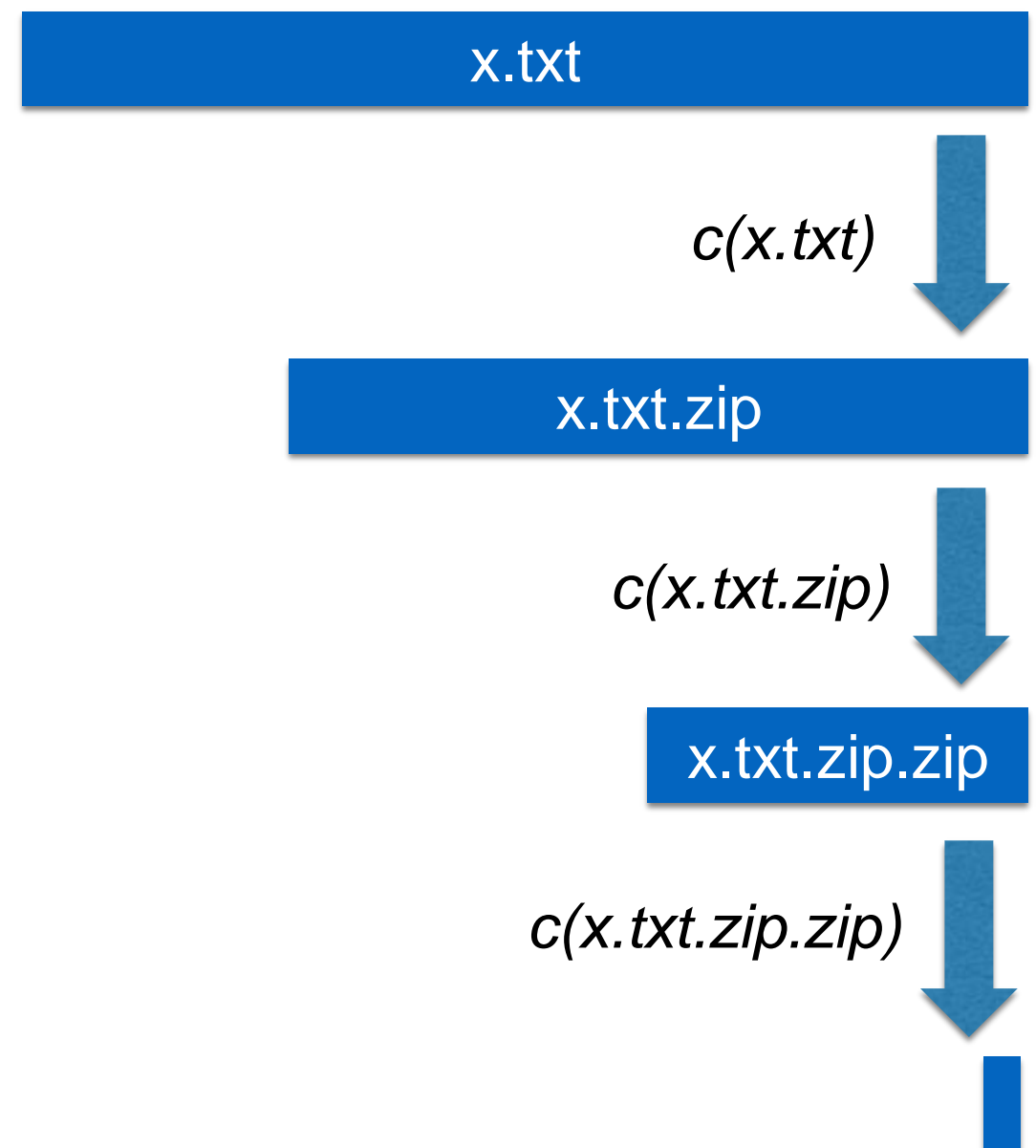
- In current world, quintillions of data is generated every day
- Issue when storing files/data on disk
- Issue when sending data on network
 - Eg, videos on YouTube, voice on Skype, etc

Compression

- Consider data as bitstring of 0/1
- Compression algorithm c from data x with length $|x|$ to data y with length $|y| < |x|$
- Need function d to decompress y into x , ie $y = c(x)$,
 $d(c(x)) = d(y) = x$
- Ie, think about Zip and Unzip commands

How Much Can I Compress???

- If c existed that always compress any input x , could just recursively apply c on its output $c(x)$ until get $|y|=1$



Compression vs. Hash

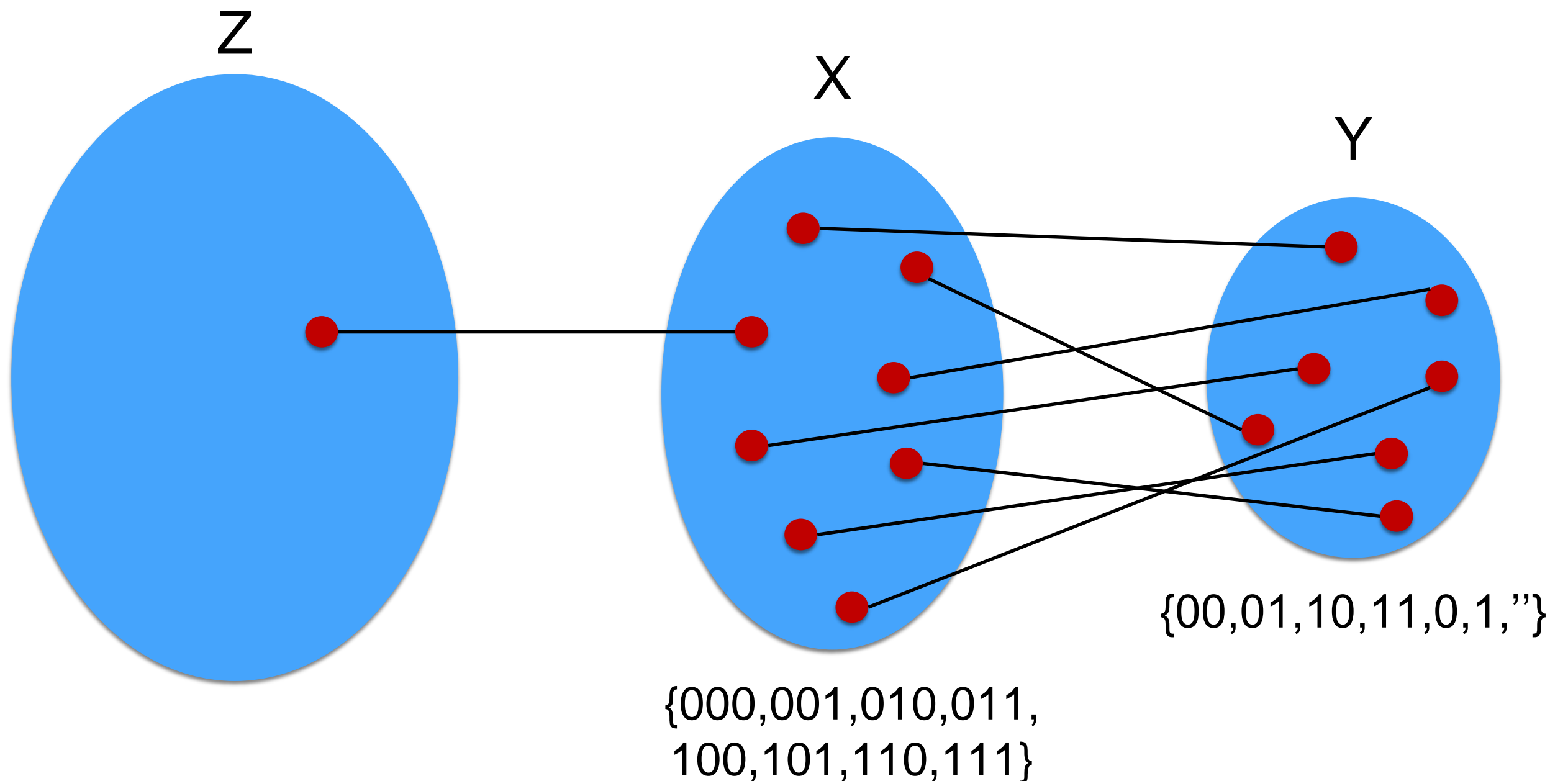
- In hash, usually we map from bigger domain X into smaller domain Y , but:
 - Should not be able to get back x from $y=h(x)$
 - There can be collisions, ie $h(x)=h(x')$ for different values in X
- In compression, still dealing with *mapping* to smaller domain Y (ie all possible shorter bitstrings), but:
 - We need function d to get back x from $y=c(x)$, ie $x=d(y)$
 - There can be no collisions, otherwise $d(y)$ would not be deterministic

Domain Size

- If X is the set of all possible bitstrings of size n , then such set has $|X| = 2^n$ elements x
- The set $|Y|$ of all possible shorter bitstrings contains the set of bitstrings of size $n-1, n-2, n-3, \dots 1$
- $|Y| = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0 = 2^n - 1 < 2^n$
- In other words, $|Y| < |X|$

Mapping $X \rightarrow Y$ and Back

- $\text{length}(z) > \text{length}(x) > \text{length}(y)$, $\text{length}(x)=2^3$, $n=3$, $|Y|=7$
- When using $c(x)$, there exists at least 1 case in which size does not decrease



What About Compression of At Least 50%?

- $|X| = 2^n, |Y| = 2^{n/2} + 2^{\frac{n}{2}-1} + \dots + 2^0 = 2^{\frac{n}{2}+1} - 1$
- $\frac{|X|}{|Y|} = \frac{2^n}{2^{\frac{n}{2}+1} - 1} > \frac{2^n}{2^{\frac{n}{2}+1}} = 2^{n-(\frac{n}{2}+1)} = 2^{\frac{n}{2}-1}$
- So, for each element x that can be compressed by at least 50%, there are $\Omega(2^n)$ elements in X that cannot be compressed by 50%
 - Eg, $n=20, |X|=1_048_576, |Y|=2047, |X|/|Y| \sim 512$
- In other words, it is *super extremely* unlikely to compress a random element from X , for increasing values of n

But, but...

- ...when I compress my files with Zip I always reduce size by quite a lot!
- Point is, you are not compressing random files, but usually files with specific structures and properties:
 - Text files (English and Norwegian language)
 - Videos
 - Etc.

Exploit Redundancy

- [illegible]

We have already compressed it...

- I did not write a trillion 1s in the previous slide
- The sentence “Consider a bitstring with one trillion ‘1’s and no ‘0’” was an instruction to (de)compress it, and it only consisted of 54 characters
- If 8 bits per character, we can compress those 1 trillion bits with just $8 \times 54 = 432$ bits
- Who is $d(x)$ here? It is the program that reads that instruction and create the trillion 1s

Another Example

- [illegible]

No Structure

- Which instruction can express the following?
- 1101100110000000011110110111000010011000
11110110010001110100110101111000 10110100
001110001101111111111110 0011011100011111
0001011010010100010001001101000110110100
00101000011101000000011011010001100010111
00010011010000110100110011100001100000100
10110111101000
- A description / set of instructions for a bitstring with no structure can be longer than the bitstring itself

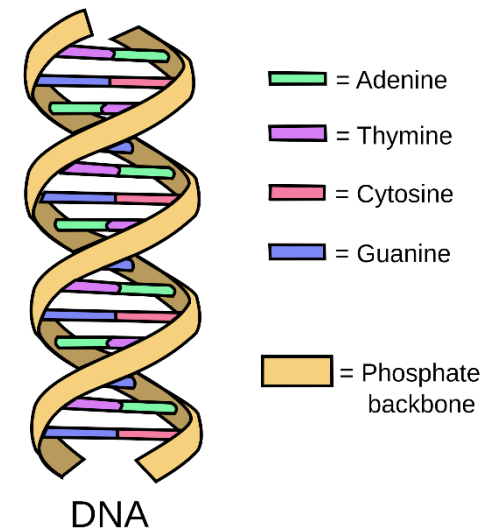
Structure in Data

- Most data you deal with has “structure”
- Eg, in English text, there is a specific set of words from a dictionary that are used, with grammatical rules, and some letters that are more common
- Usually you do not deal with compressing text like:
“Ng1LCxc7Q1a9EdkOzfV9gK3GD4DujgIcMTrpPdy
KVVtSy0Oo6U4eZG3Z3QbbTC7PRAhGUw78Yo09
3ixIf4McI9SBD483k8L1awMm”

Compression Algorithm

- Mapping function from X to Y (with an existing decompression from Y to X)
- Exploit knowledge of the domain X of the data to compress, i.e. make use of its structure (if any)
- General compression algorithm: try to automatically find structure in the data and special properties, and exploit them

DNA Data



- DNA represented as sequence of 4 types of nucleobases
 - Adenine, Thymine, Cytosine and Guanine
- How to store such data?
- Can use a *text file*, in which we use letter for each nucleobase: **A**, **T**, **C** and **G**, eg, AATCGTCGGATCGGCCCCATCG...
- Lot of data: human genome is about 3.2 billion bases
- Using 1 byte / 8 bits per character, means 3G per human genome
- Can we do better?

Finite Set

- The data in DNA sequence is from a finite set {A,T,C,G}
- Instead of 8 bits (ie 1 byte) per letter, we can use custom mapping of using just 2 bits
 - A (00), T(01), C(10) and G(11)
- So, no longer using text file, but our custom compressed format
- Going from 8 to 2 bits per nucleobase gives us a 75% compression saving

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(0010 1000
N	0100 1110	l	0110 1100)	0010 1001
				space	0010 0000

Char Representation

- Considering char encoding for just ASCII characters
- A -> 01000001
- T -> 01010100
- C -> 01000011
- G -> 01000111

Text File to Our Format

- Text: "TCGA"
- Binary of text: 01010100010000110100011101000001
- Compressed: $c(\text{"TCGA"}) = 01101100$
- Note: if I read our custom format as a text file, we would get 01101100 -> "I"

Charsets

- Before we go into the details of how to compress text files, we need to go into *more details* on how characters are represented on computers
- Each character is mapped to a bitstring representation, which can be seen as a number
- But there are many types of mappings, called *Charsets*

ASCII Codes

- **American Standard Code for Information Interchange (ASCII)**
- Mapping for 128 characters commonly used in English
 - Eg, a-z, A-Z, 0-9, ?, !, #, %, ...
- As $128 = 2^7$, we just need 7 bits, which can be store in 1 byte
- Problem: how to represent special characters like the Norwegian øæåØÆÅ, or Japanese 私はアンドレアです???

Unicode

- Standard for encoding of characters
- Representing up to $1,114,112$ possible characters
- Currently mapping 136,755 characters used in most languages around the world
- It implies we might need at least $\log_2(1,114,112) = 20.08746$ bits for the mapping, ie 3 bytes
 - ie, using a single byte is not enough

Common Charsets

- **ISO/IEC 8859-1**: using 1 byte, representing up to 256 characters, including Norwegian and Swedish ones, but not full Unicode (eg, no Japanese)
- **UTF-8**: *most used encoding*. Multi-byte representation, up to 4 bytes. Can represent whole Unicode. Ascii (most common) codes need 1 byte, but Norwegian need 2
- **UTF-16**: used internally by Java (eg, “char” variables). Each character takes *at least* 2 bytes. Covers whole Unicode.

Parsing ISO-8859-1

- As each character is 1 byte, I just read 1 byte (ie 8 bits) at a time
- Direct mapping from 8 bits to a specific character

Parsing UTF-8

- Read 1 byte at a time, but need to find out if single byte character (eg, “A”), or beginning of multi-byte one (eg, “Ø” or “す”)
- If multi-byte character, need to read all of them before being able to map them to a single char
- Look at first 2 bits in each byte:
 - 0xxxxxxx -> single byte character (using remaining 7 bits)
 - 11xxxxxx -> beginning of a multi-byte character
 - 10xxxxxx -> continuation of a multi-byte character

Generic Text Compression

- Analyze the alphabet of the text to compress
 - I.e., the {A,T,C,G} in the DNA example
- Automatically create a custom encoding of char to bits
- Idea: often used characters should have smaller bit representation than seldom used characters

Letters in *The Odyssey*

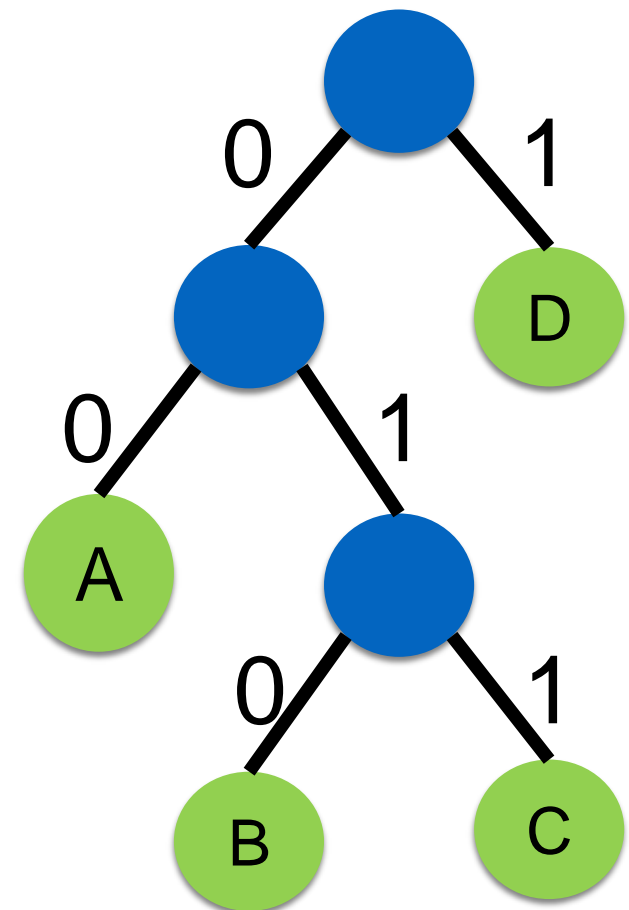
- *“Tell me, O muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy...”*
 - ' ' (108792): 111
 - 'e' (59526): 001
 - 't' (39156): 1010
 - ...
 - '1' (1): 0101001100111110010
 - '2' (1): 0101001100111110011
- Spaces and letter ‘e’ are the most common (108k and 59k occurrences), can use just few bits (ie 3) for them
 - Eg, already in opening sentence, ‘e’ used 12 times out of 110 chars
- Numbers are seldom used in that book, so use more bits (ie 17)

Prefix-Free Codes

- Whatever encoding (ie mapping from char to bitstring) we choose, it **must** be *prefix-free*
- No code should be the starting (ie prefix) of another code
- Eg, consider A=0, B=1, C=01
 - This is wrong, as A(0) is a prefix for C(01)
- How to decode 01? AB or C?
 - The decoding has to be non-ambiguous
 - If encoding is prefix-free, we know exactly each bit token to decode

Creating Prefix-Free Codes

- Using data structure called “Trie”
- Binary tree
- Labelled edges: 0 (left) and 1 (right)
- Letters in the leaves, not internal nodes
- Key for a leaf is codes on path to it
 - A = 00
 - B = 010
 - C = 011
 - D = 1
- This guarantees prefix-free mapping, as letters are ONLY on the leaves



Optimal Trie

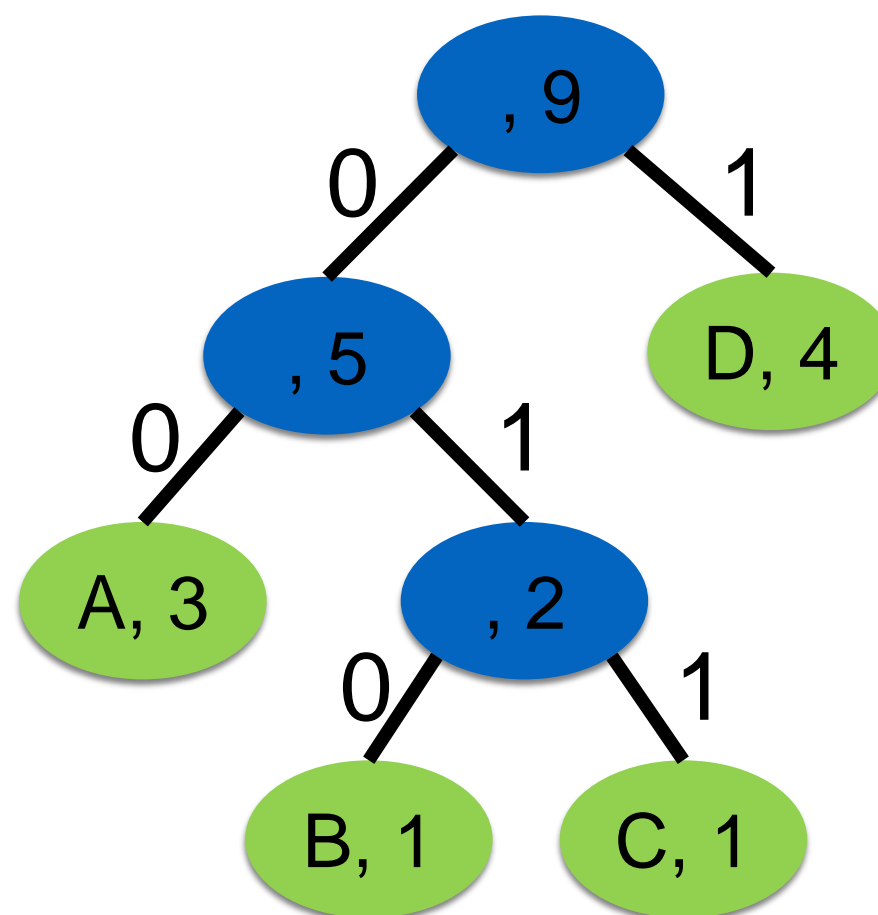
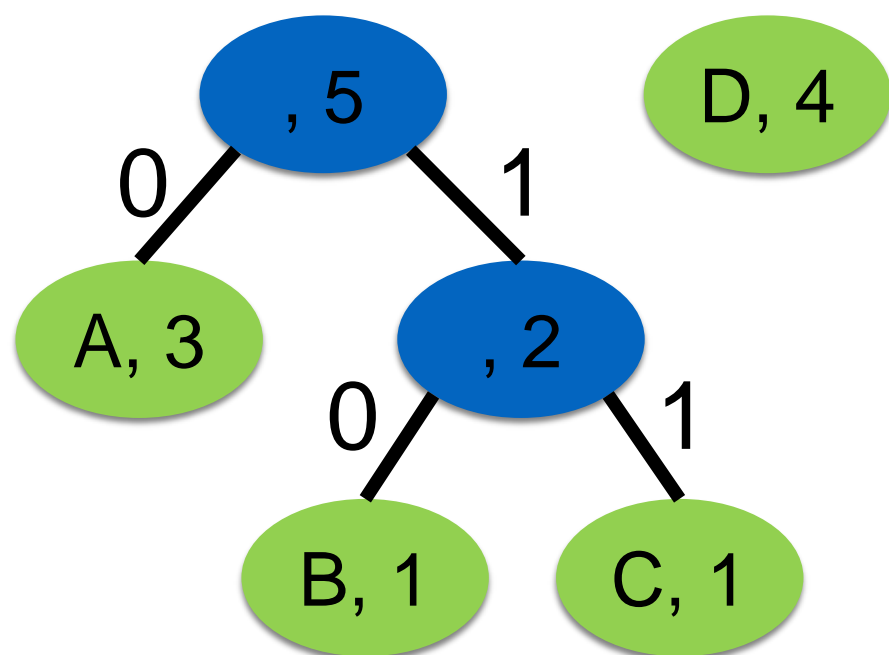
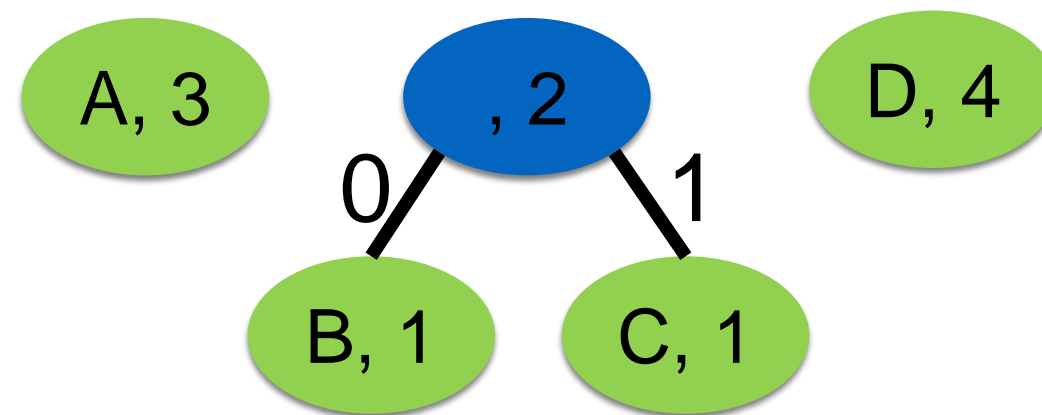
- We want most common letters having shortest bit encoding
- Given an alphabet (eg $\{A,B,C,D\}$), there can be many different tries for it
- How to build the optimal trie?

Huffman Algorithm

- Create optimal trie t for given input text x
- $c(x)=y$ \rightarrow write binary of trie t , plus x encoded with t
- $d(y)$ \rightarrow read binary of t , recreate it, use t to decode x

Creating Optimal Trie

- Consider string: “DBCADDADA”
- Compute occurrence of each symbol
 - $A = 3, B = 1, C = 1, D = 4$
- Create node for each symbol, with storing occurrences
- Choose 2 node roots with least occurrences, merge them in a new subtree with root having sum of their occurrences
- Repeat until only one root remains



Encoding The Trie

```
private static void writeTrie(
    Node node, BitWriter buffer) {

    if (node.isLeaf()) {
        buffer.write(true);
        buffer.write(node.ch);
        return;
    }

    buffer.write(false);
    writeTrie(node.left, buffer);
    writeTrie(node.right, buffer);
}
```

- For leaves: write a '1' followed by 16 bits for the char in UTF-16
- For intermediate nodes, write a '0', and then recursively write left and then right nodes
- Doing this gives us a non-ambiguous bitstring that we can decode later on to recreate the exact same trie

Huffman on *The Odyssey*

- Original in UTF-8 is 621737 bytes, ie 621kb
- Compressed with Huffman: 346507 bytes, ie 346kb
- Compression ratio: 0.55, ie we saved 45% of space

LZW Compression

- In Huffman, we have fixed size values (i.e., single chars) that are mapped by multibit key codes
- Another approach is to have fixed size key codes mapping multibit values
- Example: “*have*” is very common in English, and could be mapped by a single bitcode 000, instead of having a code for each of its chars... but need to have bitcode length n to represent all the 2^n different words/tokens in a document

End of the Course...

- LZW is a popular compression algorithm based on that approach of fixed-size codes
- Being last class of this (long) course, we will not go into its details
- But for sake of completeness, the Git repository has its source and tests, but those will NOT be part of the exam

Homework

- Study Book Chapter 5.5
- Study code in the *org.pg4200.les12* package
- Do exercises in *exercises/ex12*