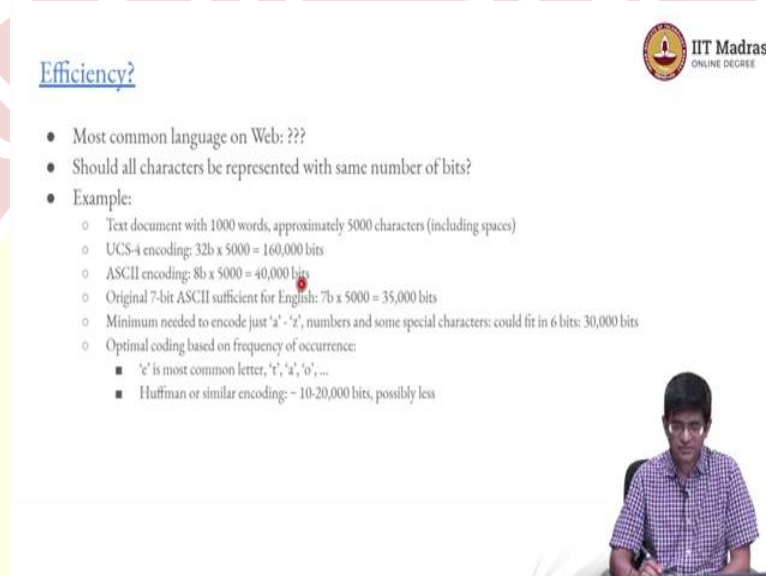


IIT Madras
ONLINE DEGREE

Modern Application Development - I
Professor Nitin Chandrachoodan
Department of Electric Engineering
Indian Institute of Technology, Madras
Efficiency of Encoding

Hello, everyone, and welcome to this course on Modern Application Development.

(Refer Slide Time: 00:16)



Efficiency?

- Most common language on Web: ???
- Should all characters be represented with same number of bits?
- Example:
 - Text document with 1000 words, approximately 5000 characters (including spaces)
 - UCS-4 encoding: $32b \times 5000 = 160,000$ bits
 - ASCII encoding: $8b \times 5000 = 40,000$ bits
 - Original 7-bit ASCII sufficient for English: $7b \times 5000 = 35,000$ bits
 - Minimum needed to encode just 'a' - 'z', numbers and some special characters: could fit in 6 bits: 30,000 bits
 - Optimal coding based on frequency of occurrence:
 - 'e' is most common letter, 't', 'a', 'o', ...
 - Huffman or similar encoding: ~ 10-20,000 bits, possibly less

So, now comes the question of efficiency. Let us look at that. First question, what is the most common language on the web? And the factor of the matter is, whether we like it or not probably just in terms of the volume, the number of web pages that are there, English is the most common language, at least now. It might be replaced by Chinese or maybe some other Indian language or something else, but it is the most common language.

And not only that, given the fact that many of the European languages use the same script, a large part of even the non-English language web would use the same set of characters or at least the, mostly the same set of characters. So, it made sense to sort of go with ASCII for a long time.

But then comes a question, I mean, obviously, you have other languages. You want to be able to represent them. When we start adding things on and as we saw that UCS-4 is good enough, 3 bytes per character is good. But I do not want to be storing 2 kilobytes when I have only 500 characters worth of information. So, let us take an example. Supposing I had a text document with 1,000 words, now, this would be approximately 5,000 characters if we include spaces

depends on what a word means. So, I am going to assume 5,000 characters as the size of my text document.

Now, with the UCS-4 encoding, 4 bytes per character, that is 32 bits per character, I end up with 160 kilobits as the size of this document. Is that acceptable? It depends on the context. But I mean, let us see if we can do better. With the ASCII encoding that would have come down by a factor of 4. So, clearly, UCS-4 is not particularly efficient. It might allow you to do a lot of stuff.

But if I know that my document is in English, do I really need to be spending this much space, because ultimately this needs to get stored somewhere on disk, in memory, and it needs to get manipulated, which means that I need to load 4 bytes every time I need to get one character of a string. Comparing strings becomes more difficult. It has many impacts that we cannot just ignore.

So, with the ASCII encoding, it would have been much better only one-fourth, 40,000 bits. The original seven bit ASCII, which would be enough if it was in English document would be only 35,000 bits, an additional sort of 8 percent improvement if you really look at it. But the minimum that is required, if we just want to encode a to z, small letters, forget capital letters, numbers, and maybe a few special characters, you could fit it within 6 bits, that would bring it down to 30,000. This is sort of artificial, because I clearly would want some kind of capital letters and so on, but at least as a thought experiment, it is useful.

Now, even more efficient encodings are possible, because from your reading of any documents in English, you would have noticed that certain letters occur more frequently than others. So, Z and Q, for example, are fairly rare occurrences in regular usage. E on the other hand is the most common. And this is pretty much well known from analysis of a large corpus of English language documents. The sequence of the highest frequency letters, E is the most then T, then A, and so on and Z, Q, J, they are at the bottom of the list.

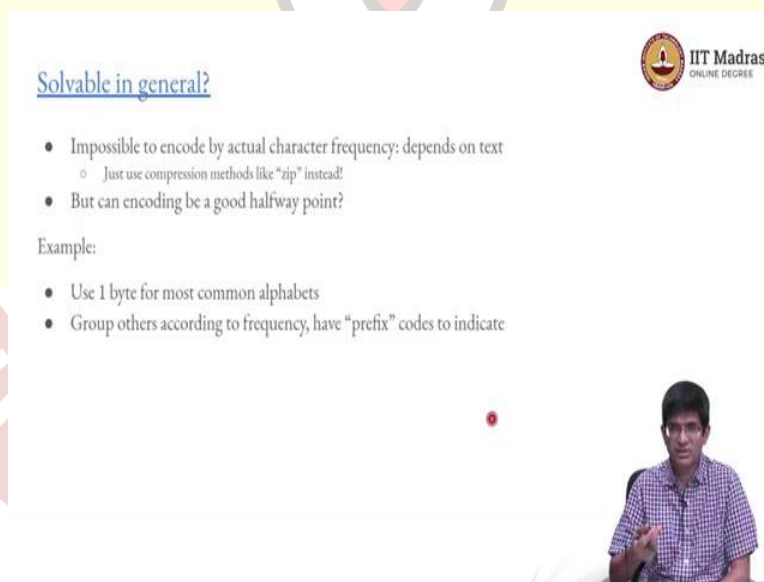
For those of you who play Scrabble, you would have noticed that the points for Z and Q and J are correspondingly higher and they are also rare. So, there are encodings, such as, for example, the Huffman encoding, which would allow you to spend fewer bits on something like E, and more bits for something like Z, because I know that E will occur more often and if I can

represent it with fewer bits, net I will get a saving and I might even be able to get by with something like 10 to 20,000 bits for this document that I have.

Is this the least that I can do? That is not a good question to ask, because now it starts to depend on the document. Supposing this document that I have said has 1,000 words basically had one word. And in fact, if that word was just A, A, A, A space, repeated 1,000 times, I could just come up with a very nice encoding which says take this particular letter A, A, A, A and have one thing which is $x \times 1,000$, repeats 1,000 times. So, that would be a total of like less than 100 bits or so in order to represent 5,000 characters.

So, the point I am trying to make is that the absolute optimal encoding depends on the document itself. What we are talking about here is independent of the actual content. Even if I do not know exactly what it is that I am trying to convey, I could, for example, do it always with a 160,000 bits using UCS-4, 40,000 bits using ASCII. But what is the best possible thing that I can do that depends on the actual content itself.

(Refer Slide Time: 05:42)



Solvable in general?

- Impossible to encode by actual character frequency: depends on text
 - Just use compression methods like "zip" instead!
- But can encoding be a good halfway point?

Example:

- Use 1 byte for most common alphabets
- Group others according to frequency, have "prefix" codes to indicate

So, the question then comes, is this solvable in general. Now, trying to encode by actual character, frequency is not always a good idea, because it ultimately depends on the text. You might have one particular piece of text which is just created to show that these characters are rare, but I am going to use them a lot. In which case, you would probably do poorly in your encoding. And the fact of the matter is what I just mentioned, trying to find out the frequencies

of occurrence of different strings or different subsets of bits and encoding them more efficiently is ultimately what any compression program like zip does.


And they are better at it, because they are doing it independent of the data. They do not really look at whether it is characters or letters. And as long as the algorithm is standardized, I can take any text document or any other document and run it through the same zip and be sure that when it comes out at the other end, it will translate into the original document, because the algorithm is now the same on both sides.

So, that helps us with the storage problem. I can always use zip to reduce the amount of data stored. But there is still the question of what about the in memory storage requirement, that is to say, if I have a document and I load it and I want to let us say compare certain strings or search for certain strings, I still do not want to be using 4 bytes per character. And then comes a question, is there a better way of encoding? Do I really need to use UCS-4 where I have 4 bytes for every character? And can we find a good halfway point?

So, as an example, let us look at is there a way by which I could use 1 byte for most of the common letters, most of the common alphabets, while at the same time saying that others, I would group them according to frequency and have some kind of a prefix code. This idea of a prefix code is something which comes from Huffman coding and related concepts.


What it tells us is that as long as there is some way to distinguish whether a given byte or a piece of information is actually supposed to represent some value or is just a prefix telling me look at the next value before you decide what this means, it means that I can now have a so called variable length encoding, some alphabets will be encoded with 1 byte, others with 2, others with 3 and still others with 4.

(Refer Slide Time: 08:23)

 IIT Madras
ONLINE DEGREE

Prefix Coding

1st Byte	2nd Byte	3rd Byte	4th Byte	Free Bits	Maximum Expressible Unicode Value
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)



And based on that, the unicode essentially came up with a kind of encoding, where they said that look, if the very first byte that you have or any byte or any 8 bit combination of bits that you find, if it starts with a 0, treat the remaining 7 bits as an ASCII code, essentially. So, you go back to the original 7 bit, original ASCII interface.

What this means is that if the very first bit is 0, you have 7 free bits to encode 128 characters and you have up to 7F that is 127 characters can be represented. All 0 is treated as a special case. It is the null byte. So, obviously, what that means is that if the first bit is 1, I need to treat this differently.

So, now, let us look at first bit being 1. Now I need another condition. The remaining part, does it start with 10 or with 11 or with something else? So, I start setting up different prefixes. If it starts with a 10, I interpret it like this, what is shown over here. If it starts at 110, I interpret it as shown in the third row, 1110 interpret as shown in the fourth row. So, each of these essentially gives us different combinations.

In other words, if I start with 110, then the remaining five bits over here and some bits of the second bite, second bite also I need to be careful. I cannot just take all the eight bits over here. Maybe I could have because, I mean, I am after all saying that I have already decided based on the first byte that this is what it is. But think about what happens if you were just scanning

through a set of bytes and you arbitrarily picked on one somewhere in the middle. You do not know if it is the first byte or the second byte, third byte, fourth byte of a given character.

So, at any point, I should be able to look at the first couple of bits and say, is this special or not? So, what that says is, if the first two bits in any byte are 10 it means it cannot be the first byte of a character. If the first bit is 0, then it straightaway means it is a simple ASCII code. If the first bit, first two bits are 11, it means that it is the first byte of a new character.

After that, we can basically say, if I have 10, 10, 10, and so on, all of these are just to indicate that these are not the first byte of a character. They are something else, which means that the x that are marked over here are all free bits and I can use them to represent different numbers. And what that tells me is, now I have 5 over here, 6 over here are, a total of 11 free bits, which means that I can go up to 2 to the power of 11, 2048 minus 1, so 2047 different Unicode values can be expressed in this.

If I have 3 bytes, then I have 4 free bits here, 6 here, 6 here, a total of 16, which gives me 65,535. And finally, I have these 3 bits 6, 6, 6. Now, over here one further restriction is what. Ideally, you have 21, which means that I could have gone up to 2 to the power of 21. But in the case of the Unicode standard, they also put in a further restriction that the first five bits essentially have to correspond to the value 10. Why? Partly, it is just a question of making it a bit easier to represent and sort of extract the values that are present over there. But you could build something else which actually allowed those bits to be anything. There is no fundamental reason why this has to be restricted.

The point is you have already hit something like a million or so different representations. So, at that point, people said, this looks like it is good enough for everything that we need. And what is the nice part about it? English language documents, for example, would fit nicely within 8 bits. Other languages, the more common ones would fit within 16 bits, less common ones would take longer.

(Refer Slide Time: 13:09)

Example

	A	א	好	不
Code point	U+0041	U+05D0	U+597D	U+233B4
UTF-8	41	D7 90	E5 A5 BD	F0 A3 8E B4
UTF-16	00 41	05 D0	59 7D	D8 4C DF B4
UTF-32	00 00 00 41	00 00 05 D0	00 00 59 7D	00 02 33 B4

See: <https://www3.org/International/articles/definitions-characters/>

Prefix Coding

1st Byte	2nd Byte	3rd Byte	4th Byte	Free Bits	Maximum Expressible Unicode Value
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)

Let us take an example. This is from the W3 website, where it talks about the definitions or characters. And we have these characters. So, this is the letter capital A from English. This is something from the Hebrew alphabet. This is something from I guess, Chinese, and or Japanese, I cannot make out the difference exactly. But the kanji script, essentially. And what we have is something called a code point. So, this is something called the Unicode code point 0 plus 0041. That is a standard. It is always the standard code point as far as this particular alphabet is concerned.

This is U plus 05D0, then 597D and this is 233B4. All of these are hexadecimal values. Now, this 233B4 tells you that it cannot even fit within 16 bits. Let us first look at the UTF-32's corresponding to these. The 0041 corresponds to just this. Literally, the first three bytes are 0, and then 41. The 05D0 gets translated into this, first two bytes of 0 followed by 05D0. 0597D similarly also gets translated this way. And this other character 233B4 gets translated into 00233B4.

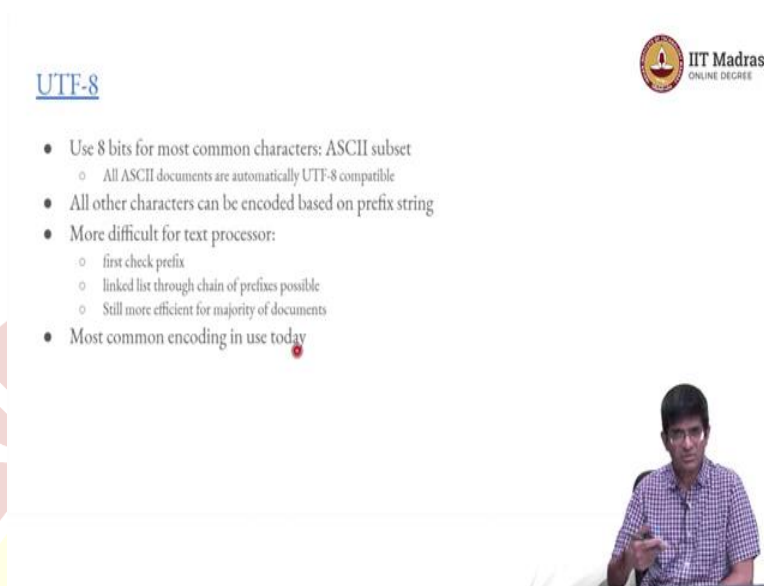
So, the UTF-32 or UCS-4 representation of these code points is very simple. It is basically the code directly converted into a 32 bit hexadecimal number, put 0 in front. UTF-8 is what I just discussed, this prefix code that we have over here. And if you go through and you work out what the encoding is supposed to be you will find that this essentially corresponds, the 41 remains the same, 41, because it starts with an MSB of 0, more significant bit of 0.

This 05D0, I need to change it, because in UTF-8 what will happen is I will need to put in this 110 over here and this 10 here and use the remaining bits in order to encode what I have. It comes out as D790. Once you put in those values, you will find that this is what it translates to. This 597D instead translates into a three byte character. And this other one, which was already 3 bytes even in UTF-32 translates into a 4 byte representation.

So, UTF-8 in other words has all of these possibilities 1, 2, 3, or 4 bytes in order to represent a single character. UTF-16 uses 2 bytes for most of them and 4 bytes for one of them. So, as you can see, both of these UTF-8 and UTF-16 are variable length encodings. Different characters will have different number of bits required to represent them.

The advantage obviously is the, you hopefully will get smaller representation in the majority of cases. The disadvantage is you now need to sort of look at each character and interpret it based on what its most significant bit and other bits are. UTF-32 is easy to sort of interpret. I just take 4 bytes at a time and figure out what it means, but it obviously occupies a lot more space.

(Refer Slide Time: 16:52)



UTF-8

- Use 8 bits for most common characters: ASCII subset
 - All ASCII documents are automatically UTF-8 compatible
- All other characters can be encoded based on prefix string
- More difficult for text processor:
 - first check prefix
 - linked list through chain of prefixes possible
 - Still more efficient for majority of documents
- Most common encoding in use today

So, UTF-8, UTF stands for the Unicode Transformation Format, and UTF-8 is something that uses 8 bits for the most common character set that is ASCII, which means that all ASCII documents are automatically UTF-8 compatible. All other characters can also be encoded based on this prefix encoding that we just talked about.

Now, keep in mind that this particular prefix encoding is not something that you need to know. You just need to be aware of the fact that there is a prefix encoding. The exact prefix encoding is determined by the UTF-8 standard and is pretty much implemented by some text processing utilities inside, which automatically know how to scan through a sequence of bytes and interpret them.

Now, obviously, like I said, this means that it is more difficult for text processors, it has to check a prefix, then it essentially is sort of doing a linked list traversal to find out, where is the end of this particular word and what does it mean. It is the most common encoding in use today. Now, having said all of this, we can come to the question of why is this important or useful to a person developing a web app. You need to keep in mind that your viewership, the people using your app might be coming from anywhere. So, if they are based in the U.S., they will typically, just be using regular computers that understand ASCII, all well and good.

So, if you just stick to ASCII, fine. But in India, for example, we will find that, not just English computers could have been set up even in the local language. So, there might be something

where even the menus and other things are set up to display in Hindi or in Tamil. In such a case, the browser by itself might be expecting Tamil or Hindi as the first language that it is going to see. And you need to have text processing capabilities internally which can handle any of these characters.

And basically saying that you are going to be using UTF-8 as the encoding for your document, it at least makes it a bit easier for the browser to say, look this is at least a standard. I know how to interpret the text or the set of bytes that are coming in from the network for this document and that I will then use whatever the built in libraries that are available for things like scanning through text, for parsing text, for splitting it up into tokens, for displaying it in different ways, figuring out which is a special character versus alphabet, all of those things will be done based on the UTF-8 encoding.

And what you will see is that pretty much most websites today have one sort of a meta tag in them, which mentions that the character set being used is UTF-8. Now, it is common enough that you should probably just say there should be the standard. Why do I even need to specify it? But it is good practice, in general, to specify it, because there are cases where people may not follow UTF-8. Why would they do that? That is not very clear.

There might be a case where they have a very limited audience and they find that spending 4 bytes per character, maybe you are in China and you do not want to spend 4 bytes per character because 99 percent of your audience is going to be using the regular Chinese character set. So, you use a more efficient encoding there. But UTF-8 is the most common one worldwide at least. And even for us in India this makes sense, because for most practical purposes, it means the maximum compatibility across different languages.