MITOCW | watch?v=xRLaQ4My3ms

[MUSIC PLAYING]

PROFESSOR: In the last lecture, we introduced an alternative set of algorithms for computation of the fast Fourier transform or rather, computation of the discrete Fourier transform. And we refer to these as the decimation in frequency form of the algorithm. As you recall, we developed the decimation in frequency form of the algorithm by essentially organizing the computation of the even numbered DFT points and the odd numbered DFT points separately and then proceeded in a similar manner.

This corresponded to breaking the computation into two and over two point DFT's proceeded in a similar manner to decompose the computation of these separately into their even numbered points and odd numbered points, et cetera. And the flow graph that resulted from this procedure was-- as I've shown here-- which is one form, then, of the decimation in frequency form of the fast Fourier transform algorithm. It begins with the data sorted in normal order and proceeds to the data sorted in big reversed order.

And as I stressed last time, it also corresponds to an in-place computation because of the way in which the butterflies are laid out. In particular, the output nodes for each butterfly are horizontally adjacent to the input nodes. Now this is similar in a number of respects to one of the decimation in time forms of the FFT algorithm.

Although, as I also indicated last time, the basic structure of the butterfly computation is somewhat different. In particular, for this set of algorithms, the multiplication by powers of W is implemented at the output of the butterfly, as opposed to the decimation in time forms, where the multiplication by powers of W are implemented at the input to the butterfly. However, there is a very close relationship between this flow graph and the original flow graph that we implemented for the decimation in time algorithm.

In particular, the two flow graphs are transposes of each other. Now I remind you that in one of the early lectures, when we introduced the notation of flow graphs and some of the properties of flow graphs, one of the properties of flow graphs that I mentioned-- although we didn't derive it-- was the fact that transposition of a flow graph doesn't change the input output characteristics. So then in fact, if we took this flow graph, changed the direction of all of the arrows, treated this set of nodes as the input nodes and this set of nodes as the output nodes, then since the flow graph, as it's indicated here, computes the discrete Fourier transform, the transposed flow graph would also compute the discrete Fourier transform.

And in fact, the transposition of this flow graph is identical to the first flow graph-- FFT flow graph-- that we introduced, which was the initial decimation in time form. And that I refer you to again, here. So this is the decimation in time form of the algorithm with the input bit reversed and the output in normal order.

This is the transposition of this flow graph, which is the decimation in frequency form of the algorithm with the input in normal order and the output in bit reversed order. The fact that these are related through transposition of the flow graph isn't a particularly important point from a practical point of view. But it is a useful piece of insight into the relationship between these various algorithms.

All right, now just as we did with the decimation in time forms of the algorithm, we can rearrange the decimation in frequency form of the algorithm to either arrange the output to be in normal order, as opposed to having the input in normal order, or rearrange it so that both the input and output are in normal order or rearrange it as we will, so that we can utilize sequential memory rather than random access memory. To rearrange the flow graph so that the output is in normal order, we would, again, simply take horizontal lines and re-sort them so that the output nodes are sorted in a normal order. And the flow graph that results is a decimation in frequency form of the algorithm, for which the outputs are in normal order.

But the inputs are in bit reversed order. Again, as this flow graph has been sorted, it is an in-place computation. In-place, again, because the butterflies are laid out in such a way that they involve horizontally adjacent nodes. So this is an in-place computation with the input in bit reversed order and the output in normal order.

Again this is a transposition of one of the flow graphs that we discussed in the previous lecture. In particular, it is a transposition of the flow graph for which, if you imagine this flow graph being transposed so that the direction of the arrows are reversed and this is the input, then it corresponds to the decimation in time form of the algorithm for which the input is in normal order and the output comes out in bit reversed order. And that is the flow graph that I have indicated here. So this is a decimation in time form of the algorithm, which is the transposition of the decimation in frequency form of the algorithm that we just looked at.

All right, that corresponds then to two possible decimation in frequency algorithms. One in which the input is in normal order, the output is bit reversed. This one in which the input is bit reversed and the output is in normal order. We can also rearrange this so that the input is in normal order and the output is in normal order.

And the flow graph that results when we implement that is similar to the corresponding decimation in time flow graph. In fact, it is a transposition of that flow graph. And just as we found in the decimation in time algorithm, for this case, although the input is sorted in normal order and the output comes out in normal order, the indexing is generally complicated as we proceed from stage to stage.

And furthermore, it does not correspond to an in-place computation, because of the fact that the outputs of the butterflies are no longer horizontally adjacent to the inputs to the butterfly. So generally, this flow graph and its decimation in time counterpart are not practically very important. A final rearrangement of the flow graph is the rearrangement which permits the use of sequential memory rather than random access memory.

And this, then, is the flow graph that is the counterpart to the flow graph that we discussed last time, which I referred to as the Singleton algorithm, or I attributed to Singleton. This is a variation of that type of algorithm, in which now, again, we recognize that the indexing is identical from stage to stage. The input is in normal order.

And the output comes out in bit reversed order. And let me just compare this for you with the corresponding flow graph that we had last time, which I've indicated here. That has the input in bit reversed order, the output in normal order. In fact, the transposition of this flow graph is the decimation in frequency form of the algorithm, for which the input is in normal order, and the output is in bit reversed order.

All right, so the indexing is identical from stage to stage. And just as we had previously, we can implement this algorithm by utilizing sequential memory as opposed to random access memory. Although the organization of the memory for the decimation in frequency form of the Singleton algorithm is somewhat different than the organization of the memory for the decimation in time form of the Singleton algorithm.

In particular, last time when we discussed the decimation in time algorithm, we had four memories, as we will this time. The first half of the data in memory A, the second half of the memory in memory B. We went, first of all, through all of memory A, then through all of memory B, storing results alternatively in memories C and D.

In this case, the strategy in utilizing the memory is somewhat different. In this case, we store the first half of the data points, again, in memory A, the second half of the data points, again, in memory B. But notice that now in the computation utilizing the computation for this point, for example, we use the first output point from memory A and the first output point from memory b.

We store the result in the first register in memory C and in the second register in memory C. Then, to compute the next two points, we take the next point from memory A and the next point from memory B, do the appropriate addition and subtraction, multiplication by powers of W, and store those in the next two registers in memory C. For this particular example, namely n equals eight, that fills up memory C.

And we proceed, likewise, accessing the input data, alternating between the two input memories and storing data first in all of one of the output memories and then in all of the second of the output memories. And then we proceed in that fashion from stage to stage. But again, one of the important aspects is that it utilizes sequential memory, for example, disk or drum or tape or shift register memory.

OK, so this then is essentially a picture of a variety of the algorithms, which can be used for the computation of the discrete Fourier transform, some of which were derived on the basis of a decimation in time argument. Some were derived on the basis of a decimation in frequency argument. But we saw, in fact, that the decimation in frequency forms of the algorithm are, in terms of flow graph notation or interpretation, transposes of the decimation in time form.

What I would like to discuss now are a few of the computational issues that are involved in the implementing the fast Fourier transform algorithm and return, also, to a brief discussion of at least one situation in which there is a preference for using decimation in time or using decimation in frequency. After we've discussed some of the computational considerations, I will then just very briefly discuss the generalization of the fast Fourier transform algorithm to situations in which the number of data points is not a power of two, but is a more arbitrary composite number. Well first of all, then, let me introduce a few of the computational issues that I would like to draw your attention to.

The first is-- and it's a point that I raised in the first lecture, in which I introduced the fast

Fourier transform algorithm-- that whereas our discussion has been directed entirely toward the computation of the DFT, it applies also, in a very straightforward way, to a computation of the inverse DFT. In particular, we know that the inverse discrete Fourier transform relationship is given by 1 over N times the sum of x of k W sub-N to the minus NK. In contrast to the discrete Fourier transform-- that is the forward transform-- which does not have the factor of 1 over N and involves multiplication by W sub-N to the plus NK, rather than W sub-N to the minus NK.

To use the algorithms that we have just been discussing for the computation of the inverse discrete Fourier transform, the modification is relatively straightforward. First of all, it involves a factor of 1 over N, which we can accommodate in a number of ways. One is, of course, by shifting the output or by applying scaling at each stage of the FFT, for example.

So this factor of 1 over N is easily accommodated. And the other difference is the inclusion of a minus sign in the exponent in powers of W rather than a plus sign for the forward transform. That means, in essence, that these coefficients are the complex conjugate of these coefficients.

Because W sub-capital N is e of the J two pi over capital N. This minus sign represents a conjugation of W. And so one procedure that we can follow for implementing the inverse discrete Fourier transform using all of the flow graphs that we've talked about is simply conjugate the powers of W that are involved in the computation.

An alternative procedure, which is basically equivalent, is to use the flow graphs as they stand, in which case the output data, which is obtained, is the desired result with little n replaced by minus n. So we can either compute the inverse DFT from an algorithm, which is aimed at the forward DFT, by conjugating the coefficients or equivalently by essentially flipping the output, modulo capital N. So all of the algorithms that we have talked about, then, relate in a very straightforward manner to the computation of the inverse discrete Fourier transform.

The second issue is that as we've seen in essentially all of the algorithms of practical interest, they involve either bit reversal at the input to the flow graph or a bit reversal at the output of the flow graph. Bit reversal, again, is a relatively straightforward thing to implement. First of all, it is important to recognize that bit reversal is an in-place computation, if we think of it as a computation.

In particular, suppose that we had the seven data indices zero through seven. And we want to rearrange this data so that the data is arranged in bit reversed order. Well that means that, of course, zero bit reversed is zero, one bit reversed is four. But of course, four bit reversed is one.

So in fact, we would want to store data with whose index is one in storage location four. But we will also want to store data whose data index is four in storage location one. So in fact, implementing the bit reversal can be accomplished in place by essentially swapping these two pieces of data.

Similarly of course, two bit reverses two. And so that wouldn't move. Three bit reversed is six. But six bit reversed is three.

And so again, we can carry out that interchange as an in-place computation. So bit reversal, then, can be implemented in place. And consequently, as we restore the data in a bit reversed order, we don't require double the storage, since we can carry that out as an in-place operation. Second of all, to implement bit reversal, of course, to implement bit reversal in hardware, to obtain a bit reversed index and hardware is very simple.

We just simply rearrange the order of the wires. To implement a bit reversed index register or an index algorithmically is a little more difficult. But in fact-- and this is discussed in some more detail in the text-- one of the most straightforward procedures, normally, is to implement a bit reversed counter so that as we proceed along with an index register accessing through an index in normal order, we can also run a counter that runs in bit reversed order and use, essentially, those two as index registers to tell us how to swap the data.

So a bit reversal, in fact, algorithmically is relatively straightforward. It's an interesting point that it is a somewhat inefficient procedure, as you'll see if you try to program bit reversal. But algorithmically and conceptually, it's a fairly straightforward procedure to implement.

A third computational issue, which I would like to draw your attention to is the question of obtaining the coefficients to use in the FFT computation. And there are basically two procedures that are commonly used. One, of course, is to store the coefficients in a table and then simply access them as they're needed.

A second, which saves storage but requires some computational time, is to generate the coefficients recursively. That is essentially using an oscillator-- programming an oscillator-- and generating the coefficients as they're needed. Along those lines, it's interesting to observe that in both the decimation in time and decimation in frequency forms of the algorithm, as you proceed from stage to stage, the powers of W that are involved in the computation are powers of a basic power of W that doubles-- or at least that changes-- in each stage as you go through the computation.

And this has implications as you think of either storing a table of coefficients and accessing them or as you think of generating the coefficients recursively. The fact that the powers of W are related from stage to stage suggests some fairly efficient procedures for doing this. One additional consideration, though, in obtaining the coefficients is that in some of the forms of the FFT algorithm, the coefficients are naturally accessed in a normal order, whereas in some other forms, they are naturally accessed in a bit reversed order.

For example, if we think of the decimation in frequency form of the algorithm, here is the decimation in frequency form of the algorithm with the input in normal order and the output in bit reversed order. Notice that the powers of W that we have here occur in what looks like normal order. At least these powers are in normal order.

And this is normal input order, bit reversed output order. Whereas if we took, let's say, the decimation in time form of the algorithm-- where we have normally ordered input and bit reversed output-- in that case, if you look at these powers of W, they're not in normal order. In fact, what they're in is bit reversed order.

So in fact, in addition to the consideration as to whether the input is bit reversed or the output is bit reversed, in some forms of the algorithm, the coefficients would tend to be stored in normal order, whereas in some other forms of the algorithm, the coefficients would tend to be stored in a bit reversed order. Or if we think about generating the coefficients, clearly it is simpler to think of generating coefficients in normal order than it is to think of them as being generated in bit reversed order. Now with regard to the question of the input and output being bit reversed, one important area in which the computation of the DFT-- in other words, the FFT algorithms-- play a role is in implementing convolution or correlation.

In that case, we compute a transform, multiply by something-- in the case of a convolution, we multiply by the transform of the impulse response-- and then implement an inverse transform. Well the fact that there are two transforms involved suggests the possibility that we can organize the computation in such a way that we completely avoid bit reversal. For example, we can choose an algorithm for the direct transform, which utilizes the input in normal order and generates the transform in bit reversed order.

We then simply have the transform of the impulse response stored in bit reversed order, carry out the multiplication, and then choose a form of the algorithm for the inverse transform, which has bit reversed input and results in a normally ordered output. So in that case, what we would have is the forward transform, normally ordered data being transformed to bit reversed data. And then as the inverse transform, bit reversed input and normally ordered output.

Well even given that, there are a number of options available. For example, we have an algorithm-- the decimation in time algorithm-- which is normally ordered input. I'm sorry, normally ordered input and bit reversed output, which we could use as our forward transform. Although as we just illustrated, it involves bit reversed coefficients.

Well we could think of storing the coefficients in bit reversed order and then using the companion decimation in time form of the algorithm, which has bit reversed input and normally ordered output, to achieve the inverse transform. However, in this case the coefficients are in normal order. So if we match up the algorithms that way, then we're faced with the problem that in the direct transform, the coefficients would be stored in bit reversed order, whereas in the inverse transform they would be stored in normal order.

Well you can ask whether for the decimation of frequency form of the algorithm, we can avoid that. But in fact, the same problem arises there. If we have the decimation in time in frequency form of the algorithm with normal input, bit reversed output, then the coefficients are stored in normal order.

But for the inverse decimation in frequency transform with bit reversed input and normally ordered output, the coefficients would be stored in bit reversed order. The solution is to match up a decimation in time form of the algorithm with a decimation in frequency form of the algorithm so that, for example, we can choose as the forward transform the decimation in frequency form of the algorithm with the input in normal order, the output in bit reversed order, and the coefficients normally ordered, which is generally more convenient. And then choose for the inverse transform not the decimation in frequency algorithm, but the decimation in time algorithm, for which the input is now bit reversed, the output is in normal order, and the coefficients are also in normal order.

So in fact, this suggests that if we are implementing a forward transform and an inverse transform, generally there are advantages to matching up the decimation in time form and the decimation in frequency form so that we have similarly accessed coefficients in both cases. Well there are, of course, a large variety of other computational issues to be considered in implementing the fast Fourier transform algorithms. Some of these are discussed in the text.

Many of them you will discover for yourselves as you try to program the algorithm. But hopefully, this discussion provides at least some indication of what some of the strategies are and some of the issues are that are involved in computation of the FFT of the forms that we've been talking about. Now all of this discussion has been related to the computation of the discrete Fourier transform when the number of data points is a power of two.

And these algorithms are referred to as the radix two forms of the FFT algorithm. As I indicated in the first lecture, in which we discussed the FFT, the FFT, in fact, is more general in that it generally applies when N is a highly composite number. We chose it to be composed of powers of two.

But in fact, there are a variety of other forms of the FFT algorithm for different radices. And in some cases, there are some advantages to be found in using not a power of two algorithm, but a different algorithm. On the other hand, in many situations, the disadvantages of that outweigh the advantages.

However, what I would like to do is conclude the discussion of the fast Fourier transform algorithm by just very briefly outlining what the structure of the FFT algorithm is more generally. And again, in the text there are a number of examples of FFT algorithms for radices other than a power of two and a more complete discussion than I feel that is appropriate to go through right now. However, let me just outline what some of the issues are in discussing a more general radix FFT algorithm.

Generally the computation of the discrete Fourier transform using this class of algorithms is directed toward capitalizing on the fact that the size transform to be implemented is a product of numbers. And it turns out that the more numbers-- the more terms in the decomposition-- the greater the efficiency that can be obtained in implementing the transform. In that case, generally one would think of these numbers as primes, since that is the biggest decomposition of any number that we can carry out.

But in fact, for the derivation, it is not a requirement that the p's be primes. So let's think of N, then, as decomposed as a product of p1 times p2, et cetera, through p sub-Nu, which we can alternatively write as p1 times q1, where q1 is then represented by the product of the remaining terms. And we can proceed, essentially, along a root similar to the decimation in time algorithms or along a root similar to the decimation in frequency algorithms.

Let me just indicate the strategy paralleling the decimation in time form of the algorithm, as we discussed that for N, a power of two. In that case, we decomposed p1 is equal to two, and q1 is equal to N over two. And we decompose the original sequence into two sub-sequences, one consisting of the even numbered points, the other consisting of the odd numbered points.

More generally, we would decompose the sequence into a set of sub-sequences consisting of every p1th point. So how many sequences are there? Well, there are p1 sequences. And the length of each sequence is q1.

For example, if p1 was equal to two, and q1 was N over two, this would be every other point. That's choosing the even numbered points and the odd numbered points. There would be two sub-sequences, each sub-sequence of length N over two.

More generally, then, if we had, say, N equal to 12, and p1 equal to three, we would generate three sub-sequences, each consisting of four points chosen by selecting every p1th, or every third point. So one sub sequence, which I've denoted here by A, would be this point and this one and this one and that one. The second sub-sequence would be the sub-sequence B, which is comprised of this point, this, this, and that.

And the third sub-sequence would be sub-sequence C, which is this point, this point, that point, and that point. All right, so we decompose the original sequence, then, into p1 sequences, each of length q1. Given that, we can organize the sum involved in the discrete Fourier transform by decomposing the sum into separate sums, involving each one of these p1 sub-sequences.

In particular, if we think of the argument p1 times r, as r ranges from zero to q1 minus one, we're selecting, with this argument, every p1th point, starting with the zero-eth point. If we think of the argument p1 r plus one, as r runs from zero to q1 minus one, we're then collecting together the terms, which are every p1th point, starting with the first point. Et cetera, we can proceed to decompose this into a set of p1 sub-sums, as I've indicated here.

Or when we combine these together, then, we can express x of k, the DFT, as a double sum, where we have the terms involving the separate sub sequences and then the multiplication by a power of W to combine the computation on these sub-sequences together to obtain the

DFT. These terms, of course, are obtained by the fact that this power of W is p1 r plus one. In the next sum, it will be p1 r plus two, p1 r plus three, et cetera.

We decompose that into a product of two powers of W. And then one of those terms-- since it doesn't depend on r-- can be removed from the sum on r and shows up in this second sum. Well I would suggest-- that's a somewhat rapid treatment-- I would suggest just simply working through that for an example.

And I think it will be clear how this original sum is decomposed into this one. All right, well now, just as we did in both the decimation in time and decimation in frequency forms of the algorithm for powers of two, we can recognize this power of W sub-N as a different power of a W with a different subscript. In particular, W sub-N to the p1 rk, as I have here, is equal to e to the j two pi over capital-N times p1 rk.

But N is equal to p1 times q1. That's the way we originally decomposed it. And the p1's cancel out. And we're left with e to the j two pi over q1 times rk or equivalently, W sub-q1 to the rk.

So substituting this for W sub-N to the p1 r k in the previous expression, what results is then x of k expressed as a sum from l equals zero, p1 minus one, of these powers of W sub-N. But then the important thing is that the inner sum involves q1 point sequences. This factor is W sub-q1 to the rk. And in fact, this entire summation is a q1 point DFT.

So pursuing this, then, what we basically decomposed the computation into is the computation of p1 q1 point DFT's. We obtained the q1 point DFT's and then combined them, according to the expression as we have here. This would then lead to the generalization of the butterfly computation as we had it for the radix two algorithms.

And if you count up the number of multiplies and adds, again, you'll see that there is some computational efficiency that results from doing this. And then, just as we did with the radix two algorithms, we can continue to decompose these transforms so that we have q1 is given by the product p2 times p3 through p sub-Nu. This product we denote as q2.

And then we can proceed to decompose these transforms-- these q1 point transforms-- into p2 q2 point transforms. Then we can decompose the q2 point transforms, et cetera. If you do this, then what you'll find-- and the counting is done a little more carefully in the text-- what you'll find is that the number of multiplies and adds involved in computing the discrete Fourier transform this way is proportional to N times a sum of the factors involved in the decomposition of N. I've indicated an additional factor here of minus Nu.

Depending on how you choose to count multiplies and adds, this factor of minus Nu either shows up or not. In fact, to make this expression consistent with what we obtained for the power of two algorithms, the minus Nu should be in here. And it's simply a question of whether you count or don't count some multiplications by unity.

Well OK, that's a very quick treatment-- discussion-- of the generalization of the radix two algorithms to more arbitrary radix. And there, as I indicated, are some examples of this, which are given in the text. Although, in fact, what you'll find is that in most practical contexts, it is a radix two algorithm that is the most efficient to use.

And I indicate, incidentally, that if you are faced with the problem of transforming data, which is of a length which is not a power of two, it of course can always be made to be of length power of two by simply augmenting the sequence with zeros. Well this concludes, then, the discussion of the FFT algorithms, the computation of the discrete Fourier transform. And there are, I hope, a number of issues which you will have an opportunity to dwell on some as you read through the text and as your attention will be drawn to in the study guide. Thank you.

[MUSIC PLAYING]