Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

L12L64_Space Complexity and its Complexity Classes

Hello and welcome to lecture 57 of the course Theory of Computation. This is also the beginning of week 12, which is the last week of this course. So far, we have seen automata theory; we saw PDAs, NFAs, DFAs, etc. Then we saw computability theory, and over the last couple of weeks, we have been seeing complexity theory. To be more specific, we have been seeing time complexity. Over this last week, in the remaining time that we have, we will briefly cover space complexity.

So, just like time, space is also an important resource when it comes to assessing how computation performs. It is also a resource that we are concerned about when it comes to computation. Space complexity is about understanding or classifying problems based on how much space they require. So, first of all, let us define what space complexity is, and then later we will see complexity classes based on space requirement, and then we will see different computational problems, where they get placed in which complexity class, how much space they require, etc.

Space complexity is given by a function, just like time complexity, where the function indicates the maximum space required by any input of that length. So, just like time, instead of maximum time required, we are saying maximum space required. And for a non-deterministic Turing machine, it is the maximum space used by any branch of computation over all the inputs of length n. So, in a non-deterministic Turing machine, it is the maximum over all the branches of computation. This is similar to how we define time complexity, but just that we are now measuring space instead of time.

But before that, the notion of time was rather well-defined. We just took time to be the number of steps. But what is space? One natural way to define or interpret space can be the following: We look at how much space is being used by the Turing machine over all the tapes. So, let's say it's a single-tape Turing machine, which means the input is only on one tape, and the input is on one tape, and that's the input tape, and the work also gets performed on the same tape.

Whatever we want, we do it on that tape. As we said, a single tape is equivalent to multiple tapes, etc. Now the question is, what is the definition of space used? So, one possibility is that, let us say, the input occupies the first n cells. So, n is the length of the input. Then, as part of the computation, we use some extra space.

So, the total space will be wherever the head accesses the tape and wherever it modifies, etc. So this, suppose this is the red and the blue part, this entire part is the space used by the Turing machine during the course of the computation. It may be convenient to assume that the space used is the number of cells on the tape. This is fine, but there is a small issue. The small issue is that now by default, the space used is going to be at least the length of the input, at least n.

The space used, by this definition, cannot be less than the length of the input. But this, in a way, is penalizing us. Just because the input is long doesn't mean that we should be using that much space. There are problems where the extra space that we require may be small. So what we should be charged for is the extra space, not really the space on the input.

But sometimes it may get messy because maybe instead of using extra space, I just modify the input and get what I want. So now it's not clear how to define what is allowed and what is not allowed. The problem here is that this model of measuring the space requires us to or forces us to use space at least the length of the input always, and we do not want that. So, we want another model where we can get a more nuanced or clearer picture of how much extra space is needed. One small point of caution here: this week, some of the presentations that I am going to do are going to be slightly different from how they are presented in Sipser.

For instance, in Sipser, I think this definition comes later during one of the reductions, but we will just stick to what we are saying in these lectures. So, I am just mentioning that there is a small difference in the way things are presented in Sipser and in these lectures. As I said, this model has an issue because there are problems that require only sublinear space. Sublinear means logarithmic of n space, like, let us say, $\log n$ space or square root n space. But then this model will force us to use at least order n space.

So, we will change the model. The new model is the following, and this model will be more accurate and more faithful in terms of how much extra space is required. We will consider a 2-tape Turing machine or a 3-tape Turing machine based on the goal of the Turing machine. So, 2-tape or 3-tape; I will tell you why 2 and why 3. So, there is a Turing machine, and the input is on a read-only tape. The input is presented and you cannot modify it; it's just presented there.

You cannot modify it; you cannot work on the input. There is a second tape, which is a work tape. All the work happens here. The work tape is read as well as write. You can write something here, you can edit here, you can modify, you can overwrite, you can read; all of that can be done.

So, this is where the work really happens. And if it is a decision Turing machine, a Turing machine that says accept or reject, then we only have two tapes. If it is a computational Turing machine, if it is a Turing machine that computes a function, let us say it has to take

x as input and compute some function on x, let us say it has to compute x^2or x^3, in that case alone we need a third tape, which is the output tape. So, if it is a function computing machine, then we have an output tape, which is a write-only tape. We want it to be write-only because we don't want it to be used for any work.

So, you just write something, and we are not going to be able to edit it or even read it. So, it's just write-only. It's a place where you write and forget it, and maybe this space will be used by some other machine or some other thing to access it. As far as this Turing machine is concerned, we just write and forget.

At the end, we want the output to be written here. So, three tapes if it is computing a function; if it is a yes-no problem that the Turing machine is answering, only two tapes. Once again, the input is on a read-only tape. We are not going to be adding anything more. The work tape is a read-write tape, and the space that we actually call the space usage of the Turing machine is the space on the work tape.

So, how much space on the work tape is used? That is the space requirement of the Turing machine. Over all the inputs of the same length, what is the maximum space used? That is the space used on the work tape; that is the space complexity. So maybe I'll just write that here: space used by the work tape, sorry, space used by the work tape is what counts towards the space usage. So now the definition is clear.

We have an input tape which is read-only, a work tape which is read-write, and an output tape if necessary, if we are computing a function. And the space usage is the space used on the work tape. So now it is clearly possible to use, let's say, $\log n$ space on the work tape. It may not be as big as n. In fact, we'll see examples where we use less than order n space.

Now let me define complexity classes. This is analogous to $dtime$ of $f(n)$ and $ntime$ of $f(n)$which we defined in time complexity. So similar to them, we have $dspace$ of $f(n)$ and $nspace$ of $f(n)$. So, $dspace$ of $f(n)$ is the set of all languages that can be decided by an $f(n)$ space or $O(f(n))$ space deterministic Turing machine. So deterministic Turing machine, that's why it's called $dspace$.

And $nspace$ is the set of languages that can be decided by an $f(n)$ space non-deterministic Turing machine. So this is $dspace\ f(n)$ and $nspace\ f(n)$. This is just like $dtime$ and $ntime$, but just analogously defined for space. And one more point is that in the textbook, they use the name space of $f(n)$. So, here I am just using $dspace$ just to kind of emphasize that it is deterministic.

So, let us see some problems and analyze how much space they require. One example is 3SAT. As we know, it is an NP-complete problem. We have variables, a Boolean formula with clauses and variables.

You have some clauses, and the formula is the AND of these clauses where each clause is the OR of three literals. Let us say x_1, x_2 etc., be the variables, L variables, and K clauses. We have already seen many algorithms, but let us see one more algorithm. We want to show that it is in $Dspace$ N, meaning a deterministic Turing machine, deterministic, not non-deterministic, that uses $O(N)$ space.

If you remember, it was NP-complete. We do not know a $dtime\ n$ algorithm. We do not know if it can be done in $dtime\ n^1, n^2, n^3, n^k$ for any constant k. We do not know if it can be done in $dtime\ n^{1000}$.

None of this is known because such an answer would settle the P versus NP question. But here I am saying I do not need n^2, I do not need n^3, I do not need n^{100}. I just need n space or O(n) space to decide 3SAT using a deterministic Turing machine. So, the algorithm is fairly straightforward. It is something that we have already seen.

Basically, we just try all the 2^n assignments. You could have a counter. So, you could have a counter something like this. Initially, it is all 0s, then it counts up, then it keeps counting up and so on.

Finally, you reach all 1s, and this counter corresponds to the assignments. This will be an L-bit counter because there are L variables. This counter will go from all 0s to all 1s, meaning all false to all true assignments. There are 2^L assignments that this counter cycles through. For each counter, for each assignment, we just evaluate the formula.

Let us say for all 0s, we substitute all 0s in the formula and see what it evaluates to. It is fairly straightforward. At each clause, you evaluate whether it is true or false. If all the clauses are true, we say the formula is satisfied. If any clause is not true, we say the formula is not satisfied.

So, for the counter, we need L bits because it is an L-bit counter. To evaluate the formula for a specific assignment, you can just look at each clause and see whether it is true or not. Some constants, maybe three, four bits of extra space may be enough, maybe five bits, at most 10 bits, right? Given that it's a 3CNF form, you evaluate a counter and see whether it is true. If it is true, you continue; if it is not true, then you can immediately say that this is not a satisfying assignment. You just need to have the three assigned values for the literals in the clause.

At the end, we accept if there is some assignment that satisfies the formula. If, after cycling through all the assignments, none of them satisfy the formula, then we reject. So, that is the accept and reject condition. We accept if there is some assignment that satisfies this. We reject if, after cycling through all the assignments, we don't find any assignment that satisfies this.

It's a fairly straightforward approach. The space requirement is surprising because here we need only L for the counter and a constant extra space. So, all that we require is O(L) space, which is L plus some constant, which is also O(n) where n is the length of the input. So, the space required is O(n). It is a deterministic linear space machine that can decide 3SAT. This is quite surprising. Forget linear; we do not have quadratic, cubic, or n^k algorithms for deterministic algorithms in terms of time that decide that. Here, in the space-bounded domain, we have a linear space algorithm. The thing is that this takes a lot of time, right? We are cycling through all the possible assignments, but we don't need much space. We are just cycling through assignments and testing each one of them. It takes time but not much space. This is one indication that space seems to be more powerful than time. We are able to do more things in the same space than we could do in the same time.

Next is palindrome. In fact, we had seen a linear time algorithm using a two-tape Turing machine, a linear time deterministic algorithm using a two-tape Turing machine for palindromes. Let us see how much space we require. Suppose the input is written on the tape and is 1,2,3,1, 2, 3,1,2,3, up to n, and the input is x. We first check the first bit, x_1, and then we check the last symbol, x_n.

If they are not the same, we immediately reject. Otherwise, we continue. Then we check x_2 and x_{n-1}, x_3 andx_{n-2}, and so on till we get to the midpoint. It's a very simple, straightforward algorithm.

You reject if any of these pairs are not equal. You accept if you don't reject until then. So, it's a very simple, straightforward algorithm. How much space is required? The only space required is to maintain this counter for your position: 1 or n or $n + i - something$ or i or whatever. So, this counter i needs to be stored.

We also need to calculate n+i-1. So, these two things need to be stored. Then we need to remember x_i, which is a single symbol, when we go to x_{n+1-i}. So, basically, we need to remember $i, n + 1 - i, and\ x_i$ and then go to x_{n+1-i}.

So, i is a number that is bounded by one, bounded from 1 to n. Right, n plus 1 minus i also is in the range 1 to n. So, all we need is $nlog(n)$ bits to store this number, sorry, to store i and n+1-i, and x_i is a symbol, so it depends on the alphabet size. So if it is binary, we just need one bit; if it is decimal, you may need four bits to store x_i. Right, so, but it is constant

because the alphabet is fixed for the input alphabet, right? So it is not variable based on input length or anything. So this requires $log(n)$, this requires $log(n)$, and this is constant.

So overall space required is just $log(n)$ space. So the overall space required is $log(n)$ space. Again, now we are seeing another instance where palindrome, which required order n time, now only requires $log(n)$space. So we are seeing that it is significantly lower space. We took SAT, which required exponential time, that we do not know of any sub-exponential algorithms for SAT. Now we could solve it in order n space, and palindrome, which we require order n time, now can be solved in $log(n)$ space.

So here we have two problems which can be done much better in terms of space. But not all problems are like that. There are some problems which require space which is of similar order. But in general, space seems to be a more valuable or more powerful resource than time. If you're given the same amount of space, you seem to be able to do more than what you can do with time, but it's not clear.

We are just talking on the basis of examples. Now, let me define some complexity classes, which are like we define P and NP, etc., in the case of time complexity. So, the complexity classes are L, L stands for log space, and NL stands for non-deterministic log space. So, log space means it is the set of all, it is actually D space of $log(n)$.

So, actually, palindrome is in L. So, it is just the letter L, which stands for $O(log(n))$ space. And NL is the non-deterministic equivalent of that. It is the same amount of space, but with non-determinism. And then we have... So notice that now in the case of time complexity, this $log(n)$ space, $log(n)$ time, etc., is not really meaningful because just to read the input, you need n; like you need to go through from the first location till the nth location.

Just to read the input requires n time. But whereas in the space, that space is not being counted, right? The space where we use to store that input is not counted. So we can actually do meaningful things with sublinear space. This is not really, usually not possible in time. You could do some simple things like there could be simple problems where you only look at a couple of leading symbols and then decide. But in general, it is not very interesting to look at complexity classes or computational problems which require sublinear time.

But in case of space, we have L and NL, which is log space and non-deterministic log space. So, palindrome is in log space or palindrome is in L. So I am just explaining why there is a log space but not a log time. So now the third thing is with respect to or analogous to polynomial time we had the class P, which indicated the set of all computational problems which can be decided in polynomial time.

So, which is n^k time by some deterministic Turing machine. So, we had P, which was the union of D time n^k; analogously, we have PSPACE. It is the set of all computational problems that can be decided in polynomial space. Right? So, this is equivalent to, or this is analogous to, P. Right? So these are some of the common complexity classes, and most likely we will not be seeing too many more complexity classes that classify problems in terms of space usage because of the time that we have. We have just this week 12 to cover space complexity classes.

But maybe we will see one or two more. So the classes that we will see are L, NL, and PSPACE. So now, having explained all of this, let us move to... So we have defined things in terms of time, things in terms of space, deterministic, non-deterministic, etc. So now let us see how these classes compare against each other. So many of them are simple, but some of them require some amount of cleverness. So we are going to see some relations between complexity classes based on time, space, etc.

So the first thing is that $DSPACE(f(n))$, for any f(n), $DSPACE(f(n))$ is contained in $NSPACE(f(n))$. We saw something similar in time. $DTIME(f(n))$ is contained in $NTIME(f(n))$ because a non-deterministic Turing machine can just act like a deterministic Turing machine. In other words, a deterministic Turing machine can be viewed as a non-deterministic Turing machine that does not use all the options.

So $DSPACE(f(n))$ is contained in $NSPACE(f(n))$. It's straightforward. And this implies that if you set f(n) equal to $\log(n)$, this implies that L is contained in NL. But one may ask at this point, is it a proper containment? The answer is we don't know. Right? Now, the next thing relates deterministic time and deterministic space. So suppose there is a language that requires f(n) time or $O(f(n))$ time to decide.

We are saying that it can also be decided in $O(f(n))$ space. This is also fairly simple. The $O(f(n))$ time Turing machine, the Turing machine that decides in $O(f(n))$time, runs only for $f(n)$ time. So, which means that Turing machine cannot use more space than $f(n)$ space because if you give 100 time steps, the head can only move 100 cells because each time step only moves one cell, right? So the space usage is not going to be more than the time, right? So, the Turing machine that decides the language in f(n) time or $O(f(n))$ time is already space-limited to $O(f(n))$ space. So, the maximum space used by the same Turing machine is $O(f(n))$. Hence, $DTIME(f(n))$ is contained in $DSPACE(f(n))$.

So, this is also fairly straightforward. Now the next thing, so instead of $DTIME(f(n))$, one may ask, what if we had a non-deterministic time Turing machine? So then we cannot use the same idea because we want to simulate a non-deterministic time-bounded Turing machine with a deterministic space-bounded Turing machine. So we cannot use the same idea. So, what we will do is something that we have already seen. If you remember, we

simulated a non-deterministic time-bounded Turing machine by a deterministic time-bounded Turing machine.

In lecture 31, while studying computability, we saw the simulation. In lecture 47, when we studied time complexity, we actually assessed or calculated how much time it takes. So, if you remember, we said that anything that can be done in non-deterministic time f requires deterministic time $2^{O(f)}$ or something like that. So, let us just revisit that. So, the same approach works here is what I want to say. So what was the approach? The approach was to, like, you have a computation tree, right? So we know that it runs in n time, not deterministic time f(n), which means all computational paths get completed in O(f(n)).

And then we want to identify if there is some computation path, sorry, some computation path that leads to acceptance. So maybe, I do not know. Some computation path that leads to acceptance. So out from the computation tree, we are looking for such a path. So what we can do is we can just go through all the paths in a breadth-first search manner.

So if you remember, we discussed this earlier. We said that each path will be labeled based on which branch it takes at each configuration. So here it will be 1, here it will be 2, and then maybe again 2, I don't know, here it is 3, whatever. So this path will be labeled 1, 2, 2, 3, something. So you have a counter that can track which computation path it is.

And we basically keep cycling through the counter. And the number of symbols in each of these positions will be upper-bounded by the branching factor. So there will be some branching factor B, which is the maximum number of children any configuration can have. And so it will be, so that will be the number of symbols each position, like each position this can take, this counter can take, right? So it's a B-bit counter or B-symbol counter or the base of this counter is the capital B. And, but then the length of the counter is no more than O(f(n)).

Because that is the height of a computation tree. So, in fact, all of this we discussed in lecture 31. So, maybe you can refer back to those lectures, but just that we did not look at it in terms of any time complexity or space complexity. So, each computation path can be encoded by a string of length O(f(n)) that is what I said, the counter, and we keep track of these strings.

And so we have two things. One is that counter. We have this counter that keeps track of the computation paths. And then we need to simulate the non-deterministic Turing machine. So suppose you fixed a number. It's a one, two, two, three something. And then for that specific string, let's say for the string one, two, two, three something, we want to simulate the NTM.

So there is no non-determinism there. The counter is already there. Based on the counter, we decide the choices. And this will require O(f(n)) space because we are just running an O(f(n)) time Turing machine. So, again, O(f(n))) time will run in O(f(n)) space also. So the O(f(n)) length counter is the first thing. And then for the simulation itself, we need O(f(n)) space, but that is for one specific counter, one specific string output by the counter.

Now, when we update to the next string, we don't need to keep track of what we did with the earliest string. We can just do it afresh; we can reuse the space, right? This is an interesting thing with the space. We can keep reusing it. We don't need to preserve what was done earlier.

We don't need to preserve everything that was done earlier. Maybe we only need to store a few bits of information. In this case, we just need to store whether we have found an acceptance earlier or not. In fact, if we found an acceptance, we would just close the entire simulation.

Otherwise, we would just continue. So, we have two things. One is O(f(n)) length counter and each simulation taking O(f(n)) space. So, overall space required is O(f(n)) So, while it is a deterministic space-bounded Turing machine that takes O(f(n)) space, which is what we have said here.

n time f(n) can be simulated by D space f(n). So this is another result. In fact, this is similar to the way we showed that SAT is in D space n. So basically we are taking a, it's kind of the same thing, but set in a more general setting. There also we explored all the 2^{power} impossible choices and said that it takes only an nnn-bit counter and constant space.

Here we are saying it takes O(f(n)) bit counter and O(f(n)) space. So, n time f(n) can be simulated in D space f(n). Next thing, we change the roles. Suppose something, so far both the results or two things we have said is D time simulating in D space and n time simulating in D space. Now, let us see how much time it takes to simulate a space-bounded machine. Suppose we have a D space machine that runs in O(f(n)) space.

And how much time does it take to simulate, right? And what we say is that it takes $2^{O(f(n))}$ time. We also need an assumption here. The assumption is f(n) is at least $\log(n)$. This is actually an important assumption. And usually, many statements in space complexity will be accompanied by this assumption. In a way, the point is that we need, we cannot really, it is not really meaningful to measure D space f(n) if f(n) is smaller than $\log(n)$.

Like if f(n) is little over $\log(n)$, D space of f(n) does not really make much sense because at least to, like one thing that we may be interested in doing is to keep track of the indices as we scan the input. So, the input length does not matter, it does not count. But let's say we need to look at the 10th symbol of the input or nth symbol of the input or n minus 10th symbol. Then we need to have a counter to keep track of where I am.

The Turing machine doesn't automatically give us a counter. We need to have a counter. So the counter will range from 1 to nnn, which means it needs log⁡f(0)n\log nlogn bits to store such a counter. So I'm saying why this assumption that f(n)f(n)f(n) is at least log⁡f(0)n\log nlogn is not, it's a reasonable assumption because below log⁡f(0)n\log nlogn, we cannot meaningfully do something. And why we need it, so this is why it's a meaningful assumption. It's not really, we are not losing anything by restricting ourselves to this assumption.

And why we need it, we'll see in this proof. So this is, for most meaningful space-bounded machines, this doesn't really hurt, this assumption. So, D space of f(n) means there is an input tape and there is a work tape. Let us assume that it is an accept-reject, it is a decision Turing machine.

And the space used in the work tape is f(n). That is what this means, D space of f(n). And let us see how many configurations. So, one thing that we can do is to run the space-bounded Turing machine. And let's see, it's a deterministic Turing machine. The left side is also deterministic, right side is also deterministic, right? And we just run that space-bounded Turing machine and see how much time it takes.

And what we are going to do is that that is going to be good enough. Just that we are going to limit the running of the time-bounded Turing machine to this much time. And we are going to say that we will always get a decision. So, we just run the same machine and if we measure the time taken, it is going to be at most $2^{o(f(n))}$.

If it does not conclude by that, we can stop the computation and reject the input. So, let us see why that is the case. So, the claim is that if something runs in D space f(n), the time taken is $2^{O(f(n))}$. So, let us see why that is the case. So the number of configurations of this space-bounded Turing machine, let's see what it is.

So we have multiple things which constitute the configuration. One is the head positions. There are head positions which are in the input tape and the work tape. So this is because of head positions. So the input tape, the head could be in anywhere from n positions.

n is the length of the input. And the work tape, the length is f(n). So the head position can be anything from 1 to f(n). So there could be f(n) possibilities for the head position of the work tape and n possibilities for the head position of the input tape. And the input tape contents are fixed for a specific input. So, there is no scope of changing. What is the number of possible contents of the work tape? So, contents of the work tape, it is going to be γ^(f(n)).

So, contents of the work tape, it is going to be size of γ^(f(n)), where γ is the tape alphabet. So, contents of the work tape because there are f(n) cells and each of these cells can have any tape alphabet, tape symbol there. So, it is γ^(f(n)). And then we have the state, the number of states which is q. So, these are the things that comprise the configuration.

Now, if we just see this, we can simplify as 2^(some constant times f(n)). So, we can see γ is 2^(some constant). And we have n f(n) here and q here, it is the same thing. But then all of these things are dominated by the 2^(constant f(n)) term, n f(n) q. So, q is a constant, n is a linear term and f(n) is going to be, we know that f(n) is at least log n.

So, it is going to be dominated by 2^(f(n)). So, what we can do is we can easily absorb all this into the exponent term. So, we can replace it with 2^(another constant f(n)). So, 2^(c2 f(n)), but it is just another constant which we can say 2^(O(f(n))). So, now the number of configurations of the space-bounded Turing machine is 2^(O(f(n))). Now, number of configurations is 2^(O(f(n))) means it has to terminate in 2^(O(f(n))) steps. If it does not terminate in 2^(O(f(n))) steps, that means it is repeating some configuration, which means it will just get stuck in some loop.

So if you run in that time, and if we accept, fine, we accept. If we reject, fine, we reject. But if it does not terminate in 2^(O(f(n))) steps, 2^(O(f(n))) steps, that means we are stuck in some loop. Because there are only, let's say there are only 100 configurations. And if you have run 110 steps and we have still not accepted or rejected, it means that some configuration is repeating and it is just going to repeat again.

We are just basically stuck in a loop. So, there is no point in just going through the loop over and over again. We may just as well accept that it is stuck in a loop and reject the output, reject the input. So we just run the DSpace machine and limit the running by this many steps, 2^(O(f(n))) steps. If it accepts, we accept. If it does not accept, in that much time, we reject. So the main key idea here is that we bounded the number of configurations, and that gave us an exponential time bound in the space complexity.

And obviously, this gives us a corollary. So, if f(n) is set to be log n, if f(n) is set to equal log n, we get that if we set f(n) to be log n, we get that DSpace of log n is obviously L and DTime of 2^(constant times log n) is obviously P or some polynomial. So, we can say it is contained in P. So, L is contained in P.

So, this is something that may not be too surprising. It is something that runs in log space will not take more time than polynomial time. So, here we compared the deterministic space and how much time it takes to run a deterministic space Turing machine. The next thing is we will move to a non-deterministic space Turing machine. So, we move to a non-deterministic space-bounded Turing machine. Again, we use the same assumption that f(n) is at least log n. One point, we actually use this assumption over here, right? We use this

assumption over here because if we don't have this assumption, we cannot assume that n f(n) is dominated by 2^c f(n), right? Because you assume that n is dominated by 2^c f(n).

So, we have to assume that f(n) is at least log n. Otherwise, we cannot do that. That is where this assumption actually plays a role in this proof to move from basically this inequality. So, similarly, it will play the same role here as well. So, now we are saying that instead of a deterministic space-bounded Turing machine, we are dealing with a non-deterministic space-bounded Turing machine, but the result is that it takes the same amount of time. It can be simulated in d time 2^n. So, this right-hand side is the same, but the left-hand side has been replaced by n space. We cannot use the same simulation that we did here, where we just ran the d-space Turing machine because d-space Turing machine is just deterministic.

But non-deterministic space-bounded Turing machines could have multiple approaches. So, now we cannot run it just like that, because running it itself requires us to explore many computation paths, etc. And if there is a branching factor of B, then if you do some kind of exploration of all paths, it may result in some kind of double exponential thing like B^2^O(f(n)) configurations, because we have a tree that is of height f(n). Sorry, we have 2^O(f(n)) configurations. So, the tree could be as big as 2^O(f(n)) size with that many different configurations, but then each configuration can have as many as B children.

So, this could be as bad as this. So, we do not really use this kind of straightforward approach. The thing is that the total number of configurations is bounded, right? So we already saw that the total number of configurations in the case of a d-space Turing machine is bounded by 2^O(f(n)) and n-space also has the same bound because the configuration doesn't really depend on whether it is deterministic or non-deterministic, right? It is just contents of the work tape, head positions, and state, right? So the number of configurations is going to be 2^O(f(n)), right? Or at most 2^O(f(n)). Right? So, the point is that many of these B^2^O and things, the configurations, they are like duplicates. Like, the same configuration may be appearing in multiple computation paths.

So, we do not need to duplicate the effort. Instead, we can just look at the entire tree, the entire set of configurations as a single directed tree. Right? So, we have the starting configuration of the non-deterministic Turing machine on input. So, whatever input it is, once we decide the input, the starting configuration is fixed. And then how many valid successors does it have? So, some successors and they will have more successors, and so on. So, we will get some kind of a directed graph which is a configuration graph.

And we just want to know whether there is some way to reach the accepting configuration from the starting configuration. But then there is another problem here. The problem is that there could be multiple accepting configurations because the accept state is unique, but the

tape contents could be different. But what we can do is modify the Turing machine such that whenever it accepts, it erases the tape contents.

So, it erases the tape contents and brings the head back to the left end, which means there is only a single accept configuration. We can assume there is a single accept configuration. So now, in the configuration graph, so you start from the, so it's a directed graph where you have an edge. You have an edge if there is a valid successor, if this configuration is a valid successor of the previous configuration.

So you have a directed edge from i to j if j is a valid successor of i. And that is easily checked by the rules of the non-deterministic Turing machine. So we can build this configuration graph. Another point is that we do not need to explicitly build it. We can kind of have this idea in mind and just build it as we guess or make the next step as we go along. We can kind of on the fly check whether the edge is there or not.

All we need is access to the information whether there is an edge from i to j, and this can be done on the fly. We don't need to set up an adjacency matrix because that's again going to be too much space and too much time. So we don't need to do that. In fact, in this case, we can actually build the configuration graph and build the adjacency matrix also because anyway, the running time that we are using is 2^O(f(n)). So we can build this configuration graph and we need to check whether there is a path from start to accept.

So we can use any traversal algorithm, right? We can just use breadth-first search or depth-first search, which runs in linear time in the size of the graph. So v plus e time, which is upper bounded by v^2 time. And we know the number of configurations is 2^O(f(n)), right? Something that we said here, as well as we said here, right? So that implies that the running time of the BFS is also going to be 2^O(f(n)).

The square gets absorbed in the O here. So it's just asymptotics. And even this O is unnecessary. So the running time is 2^O(f(n)). So what we have shown is that an n-space Turing machine, an n-space Turing machine which runs in O(f(n)) space can be simulated in d time 2^O(f(n)). Right? So, again, even though it is a non-deterministic space-bounded machine, it took the same time as a deterministic space-bounded Turing machine, right? So, this is also d time 2^O(f(n)), this is also d time 2^O(f(n)), right.

And this implies that by setting f(n) to be log n, we get that you set f(n) to be log n, we get that n-space of f(n) is NL. So, n-space of log n is NL. And d time of 2^O(log n) is basically d time of n^constant, which is polynomial time. So, this shows that NL is contained in P. So, yeah, I think that is what I want to say in this lecture, lecture number 57. So we introduced space complexity, which is the maximum space taken by an input of any length.

And in the case of a non-deterministic Turing machine, it's the maximum space used by any branch of computation on any input of the same length. And then we said that the

standard model of computation, the standard Turing machine model may not be good for measuring the space because it forces the space requirement to be always at least n. So, we considered or we structured another model where we have the input separated from the work tape, where the input is read-only and you cannot work on the input tape. And the space that counts is only the space in the work tape.

Now, with this model, we can make meaningful statements like this problem can be decided in O(log n) space. So, then we said some statements. Statements, so we defined some complexity classes like L, NL, and log space, sorry, NL, NL, and P space. So L is log space, NL is non-deterministic log space, and P space is polynomial space. And we saw some relations versus the D-space f(n) is contained in N-space f(n).

D-time f(n) is contained in D-space f(n). So one consequence of this is that P is contained in P space, polynomial time and polynomial space. Yes, and then we saw that non-deterministic time is also contained in the same space. So, this shows that even NP is contained in the P space.

That is what this shows, right? And then we saw how space-bounded Turing machines can be simulated by time-bounded Turing machines. So, d-space of f(n) is contained in d-time 2^O(f(n)) and n-space f(n) is contained in d-time is also contained in d-time 2^O(f(n)). So, by setting f(n) to be log n we get that L is contained in P and even NL is contained in P.

So, that is some introduction to what is space complexity and also some relations between how these space complexity classes relate with each other as well as with time-bounded complexity classes. And in the next lecture, we will see something called, just like we had NP completeness, we will see a notion of completeness, but it will be NL completeness, not P space, but NL completeness. So, that is what we will see in the next lecture. As far as lecture number 57 is concerned, that is all from me and see you in the next lecture.