welcome to the course on design and analysis of algorithms the topic for today is approximation algorithms for NP complete problems
so let me start with a question suppose we have an NP complete problem which we need to solve so you wanted to solve a certain problem which arose in some real life situation and it turned out that it was NP complete what you do (refer slide time: 03:28)

this is going to be the subject of the next two three lectures how do we cope with the problem which is known to be NP complete
usually NP complete problems arise when we talk about optimization problems so if finding an optimal solution is NP complete one wonders whether we can at least get nearly optimal solution in polynomial time.

This approach is actually quite promising and it’s the approach of finding fast approximation algorithms it will be we will not be interested in this in this two three lectures in finding the optimal solution but we will be interested in finding an approximately optimal solution and therefore we will be devising algorithms which are called approximation algorithms.

And our hope is that for the real life application that we are worrying about the approximately optimal solution that we find is also fairly useful.

Another possibility is to examine whether the real life problem that we want to solve has additional features that make it a special case of an NP complete problem if this is true then sometimes the special cases can have efficient algorithms can have polynomial time algorithms.

So for example vertex cover is NP complete but if you are finding about X cover on a tree or on the bipartite graph we do have fast algorithms polynomial time algorithms for solve for solving such problems.

So it’s useful to think about whether the real life problem that we are solving has any special features sometimes those special features can be exploited to get fast algorithms.

Another possibility is to find what are called pseudo polynomial time algorithms so let me explain this a little bit more.

An algorithm is said to run in pseudo polynomial time if its run time is polynomial in the size of the input instance so for so good so far like the usual definition here is a difference the run time is polynomial in the size of the input instance when the numbers in the input are represented in unary in the normal definition or in the definition of polynomial time we require that the numbers be represented in binary or in some radix which is larger than one.

What happens if you represent them in unary so just to clarify if we have a number thirteen that we represent that we want to represent
in the unary number in the unary representation system it will be represented by a string of thirteen ones
so note for one thing that this representation is going to be much longer than this representation
if you look at binary this is going to be the representation but this is still substantially smaller over here over than over here in fact the number of bits needed over here is log of this
so there is going to be a bit difference between the length of your input instance when measure it in unary or in binary (refer slide time: 08:35)

so naturally if you are only interested in devising algorithms which run in time polynomial in the length of the unary representation you have got a lot more freedom to work with ok your algorithms can take somewhat longer than if they where to be running in time polynomial in bi in their binary representation when the numbers are represented in binary
we have in fact seen a pseudo polynomial time algorithm in this course so this was the
naples the dynamic programming algorithm that we saw for the knapsack problem
[sneezing]
let me remind you what the problem was you are given \( n \) items specified by their weights
and values and you are given an integer capacity for a knapsack all these where integers
this description applies only twenty ((05:51))
now it was assumed implicitly that the weight value and capacity are in \( d \) bit numbers so
since there are two \( n \) weights and values and one capacity the input size is \( d \) times two \( n 
\) plus one
and if you remember we showed that the time taken for the knapsack problem was \( O \) of \( C 
\) times \( n \) where \( C \) is the value of the capacity so this is crucial its not necessarily the length
of the bit string needed to represent the capacity but its actually the value of the capacity
[sneezing]
now if these numbers are represented in unary then the time taken is \( O \) of \( d n \) because then
\( C \) would be \( d \) bits long ok so the so this \( C \) itself would be \( d \) as would be smaller than \( d 
\) and therefore there is no problem the time taken would be \( C \) times \( n \) but \( C \) times \( n \) is also
\( d \) times \( n \) is at most \( d \) times \( n \) as well
and this is certainly polynomial in the input size because the input size is just this in fact
its linear in the input size [sneezing]
however if the numbers are represented in binary what happens well if the capacity is
represented as a \( d \) bit binary number then \( C \) can be as large as two to the \( d \) so then this
time \( O \) of \( C \) times \( n \) really could be as large as \( O \) of two to the \( d \) times \( n \) (refer slide time: 
09:44)
now notice that this expression two to the d times n is not polynomial in this expression
whatever power you take of this whatever constant power you take of this you cannot
beat this [sneezing] and therefore this is not polynomial
so clearly polynomial time if you can get polynomial time its better than pseudo
polynomial time ok so pseudo polynomial time is not the the best possible ok or its its
really different from our notion of good algorithms which are polynomial time algorithms
however pseudo polynomial time is better than exponential time so that’s also worth
noting because the length is O of nd ok and exponential would be something like two to
the power nd so here we are getting d in the exponent but we are getting n not in the
exponent n just as a multiplier
so this is certainly still much better than this so as a compromise between polynomial
time and exponential time it is useful to think about whether there are pseudo polynomial
time algorithms possible for the problem that you want to solve
then people do look at algorithms which are difficult to analyze but instead of analyzing
them they try out lots of instances and check whether the algorithms run fast enough this
is what i mean by saying alg we try to discover algorithms which work well in practice
such algorithms are called heuristics
and they do tend to be useful when solving problems which are known to be very hard so often it it may turn out that you may have a good heuristic which you really cannot analyze but it seems to do the job if it seems to do the job why not yourself [sneezing] the last idea which is also used is to use the exponential time algorithm so if a problem is NP complete we know that it can be solved by an exponential time algorithm so we use that exponential time algorithm if the problem size is small or small enough then the time taken may be acceptable or if if the problem is just to be solved once then even if the problem takes a day doesn’t matter we will run a computer for a day and get a solution so this also works sometimes sometimes for solving real life problems the real life problems tend to be reasonably small and today computers are getting really fast so exponential time algorithms can work its not that they are entirely useless our focus at these lectures however is going to be on approximation algorithms we would like to device algorithms which are provably fast ok which are ((10:31)) in polynomial time that’s all that we mean in this in these two three lectures when we say provably fast that there are in polynomial time and while they may not give the optimal solutions we will prove that they will give somewhat close to optimal solutions
so here is the outline for today so i am going to define the notion of approximation algorithms i will also define a term called the approximation ratio of an algorithm or an approximation factor of an algorithm and then i will describe approximation algorithms for two problems one is the metric the metric traveling salesman problem and another is the precedence constrained scheduling problem so lets begin with the definition of approximation algorithms so lets say p is denotes an optimization problem p is an optimization problem and it look something like minimize this objective function subject to these constraints of course it could be maximize but for definiteness lets consider minimization first let A of i denote the cost of the solution found by an algorithm A on instance i ok so we are not worrying about the time right now ok we are worrying about the objective function cost (refer slide time: 14:54)

so we want this objective function cost to be as small as possible but this algorithm A when run on instance i produces this objective function value suppose OPT of i denotes the cost of an optimal solution to this instance i for technical reasons will assume that OPT of i is greater than zero we will see why in a minute
now we define the approximation factor or the approximation ratio \( \rho \) on instance \( i \) as \( A \) of \( i \) upon \( \text{OPT} \) \( i \) what is the factor by which \( A \) is worse than \( \text{OPT} \) \( i \) that is what this approximation ratio is all about so it’s a natural definition clearly \( A \) of \( i \) the cost found by the algorithm can at best be as small as the optimal cost in general it could be larger and therefore this \( \rho \) sub \( i \) is going to be larger and we would like it to be as close to one as possible in general the approximation factor of this algorithm is just the maximum value of \( \rho \) sub \( i \) over all possible instances of size \( n \) so its customary to use the worst case by enlarge and so here as well we are going to look at the worst case ratio and of course its going to be parameterized by the size \( n \) so we will write this as \( \rho \) of \( n \) so for different \( n \) we will have a different ratio so in fact we are looking for looking to evaluate this and we are looking to keep this small the goal clearly is to design approximation algorithms or algorithms such that \( \rho \) of \( n \) is small as close to one as possible for large \( n \) and of course the time for this algorithm is polynomial the algorithm must run in polynomial time (refer slide time: 20:38)

sometimes you want to maximize the objective function in which case we will define \( \rho \) sub \( i \) the approximation factor as the reciprocal
so now we know that A of i can at most be as large as this and therefore it will turn out
but this is still going to be bigger than one
so again our goal is going to be similar ok so rho of n is going to be the same and the goal
is also going to be the going to be similar we want appro we want algorithms which keep
rho of n as close to one as possible [sneezing] which get rho of n as small as small as
possible
so now we will use these ideas to device an approximation algorithm for the metric TSP
problem
so let me define this problem first the input to this problem is a graph G and this is going
to be specified as an n by n metrics D in which D of i j denotes dis the distance between
vertex i and vertex j in this graph
now the metric in the in the title says that D has to have certain additional structure
specifically D has to form a metric and by that we mean first of all for all i the distance of
a node to itself is going to be has to be zero
the distances have to be symmetric the distance from i to j has to be the same as the
distance from j to i [sneezing] and the final thing is that for all i j k the distance of going
from i to j directly has to be no larger than the distance of going from i to k first and from
k to j next this is often called a triangle inequality constraint (refer slide time: 26:04)
so imagine that i j k are the vertices of a triangle and this just says that the straight
distance going from i to j is smaller than the indirect distance
so let me first take an example of what a metric problem is going to be ok so i am not
going to draw the metric the metrics D but i am just going to take the problem and i am
going to draw the graph corresponding to the problem
so one way to use such a graph is to is to imagine that the vertices are embedded in the
Euclidean plane
so for example here is vertex one here is vertex two here says vertex three here says
vertex four i could draw out all the edges but even without drawing all the edges let me
tell you that the distance from i to j is simply the straight line distance in the plane
so D of i to j is straight distance straight Euclidean distance in the plane now all of us
know that the distance from here to here plus the distance from here to here can never be
smaller than the shorter distances from the straight distance and so clearly our third
constraint the triangle inequality constraint is obviously applicable over here
so for completeness i could write down this this is the graph that here looking at ok for
example and if you do the arithmetic you could say for example that you could calculate
the distances ok so this is the graph
and if you look at D ij to be the Euclidean distance then clearly it will satisfy all these
metric property the properties mentioned over here so this would be a traveling salesman
problem instance ok what is suppose to be output
what is suppose to be output is a cycle in the graph passing through all vertices exactly
once such that the sum of the distances associated with the edges in the cycle is small as
is as small as possible
so this is the same thing as in the TSP problem you want to tour you want to tour in the
graph passing through every vert every vertex such that the tour length is as small as
possible
the first claim is that metric TSP is NP complete well i think we have studied earlier the
TSP is NP complete but it turns out that even with these restrictions so this is the special
instance of a TSP but even with these special restrictions TSP remains NP complete
here is the claim that we are interested in and which we are going to prove so the claim is
there exist a two approximation algorithm for metric TSP here is a quick overview of the
proof in fact the proof is actually quite simple the idea is actually quite interesting and but short
so the general idea is this and this scheme appears in other places as well so first we are
going to find the lower bound L on the length OPT of the optimal tour so whatever graph
we are given it has some optimal tour will try to figure out a lower bound on it
then we will construct a tour of length C which is at most twice this so notice that it is
very hard to figure out the length of the optimal tour
so we really want a tour which has say twice the length of the optimal tour but rather than
that we will find a lower bound which which will be easily computable and we
will show that we can construct a tour which is at most twice the length
but since this is a lower bound we know that this is that C must also be less than twice
OPT because L is less than OPT therefore C is less than twice OPT so this is going to be
what we are going to do
so we will look at each step in turn so this is the first step so we want to find a lower
bound on the length of the optimal tour
so here is the main claim the claim says that the weight of a minimum weight spanning
tree of G with D as the weight matrix is a lower bound L on the length of an optimal tour
of G (refer slide time: 23:12)
so this is the lower bound that we wanted so i just have to prove this so lets imagine that we are given any optimal tour we take that optimal tour and we remove an edge in it edge from it what do we get well we will get a path which passes through all the vertices of the graph once it starts at some vertex and it passes through all the other vertex and returns to some other vertex but is there anything interesting that we can say about this path well this path is also a special case of a spanning tree this is a spanning path it passes through every vertex and therefore this also is a spanning tree of G so its length its total length is certainly no smaller than the weight of the minimum spanning tree because the minimum spanning tree is by definition that spanning tree whose weight is the least and therefore the length of this which is the weight of the corresponding the spanning tree by the way in this case weight and length are to be use synonymously weight is the terminology used in connection with minimum spanning trees and length in connection with tours so i am sticking to those spi spick stick into that but really length and weight are the same so the length of the path has to be greater than or equal to the weight of the minimum spanning tree it will be equal if the path itself happens to be the minimum spanning tree
but the length of the tour is bigger than the length of the path because the tour in fact contained an extra edge and therefore the length of the tour is also bigger than the weight of the minimum spanning tree but this minimum spanning tree has beat L and therefore we are done ok

so the length of the optimal tour is bigger than L so we have proved this we have established a lower bound L on the length OPT of the optimal tour

now we want to argue we want to construct a tour and argue that its length is at most two times L and once we are done we will have proved our result [noise]

so here is have you construct a tour with length less than two times L so i am going to give you the algorithm

so first we find a minimum spanning tree which allows us to determine the L so the weight of the tree is L we can actually write this down we can find the minimum spanning tree and we can find its weight and that is going to be L the lower bound

next we do a DFS a DFS traversal depth first traversal of T or do a depth first search of T and we look at the sequence of vertices that get visited and that sequences return out as E so lets take our graph and lets look at that sequence so here is our graph now we start at one and if we are doing the depth first search well there could be many ways in which we do the depth first search ok

so first of all i have to identify what this tree T is going to be so clearly this is going to be the tree T so this is going to be the minimum spanning tree in this graph the red edges

now if i want to do a depth first traversal of this tree say starting at vertex one what would i get so from here let me just use red again (refer slide time: 25:24)

so from here i would visit two then from here i would visit three then i would go back then i would go forward then i would go back then i would go forward and so the E that i get is going to be something like this so i start with one then i go to two then i go to three then i go to two then i go to four then i go to two and then i go to one

so this is going to be my sequence E so one two three two four two one ok so this is how i have constructed E

now the algo the idea is that if D appears more than once in E so there are several vertices which appear more than once we are going to delete its first appearance ok
so one appears more than once but this one is really the same as this one because the tour
tour is just closing so we don’t worry about this so the first that appears more than once is
this two
so now we are going to delete it and we are going to replace it by the direct edge ok so
we are going to delete one to two and two to three and we are going to replace it by a de
by a direct edge ok or may be i will use black this fine this will be perfectly
understandable
and then we are going to repeat the previous step while possible so if a vertex appears
several times we are going to short circuit it we are going short cut it ok so our current E
now is going to be this we have just removed this ok
so the next vertex that appears twice is two ok it already appeared but it is going to be
two it could be another vertex but in this case it just happens to be two ok
so we are again going to delete its first occurrence so if we delete its first occurrence
what does it mean instead of going from three to two and two to four we are going to
remove these edges and we are going to replace it with this direct edge (refer slide time: 27:41)
so we are going to keep on doing this step as many times as needed ok and at the end we
are going to return E ok so let me draw another picture to show you what this E that was
that is to be return is
so this is our vertex one vertex two vertex three vertex four so in our so original tour was
one two three two four two one ok
so we removed this two and then we removed this two and we were left with this so our
our E that remains at the end is going to be this (refer slide time: 28:58)
so going directly from one to three then going directly from three to four then four to two and two to one so this is the E that we would be returning this is the claimed final answer

ok

so lets go over each step and we will ((29:09)) about how exactly it is done so the first step is done as is finding the minimum spanning tree ok

so how long does it take well if we use prims algorithm it will take something like E plus V log V so clearly polynomial time how long does this take this is just depth first search so it takes time linear in fact so this time is less than this time [sneezing]

so now let us worry about so these steps are again going to be fairly straightforward so lets not worry about the time but lets worry about the correctness

so once we find this E ok we eventually we modify it and eventually return E so lets try to device lets try to figure out some properties of E (refer slide time: 30:57)
so my first claim is that weight of E after the step two is going to be twice L so that actually is obvious from this picture but i will just draw it again
so our graph was this and our tour was this so notice that our tour used every edge our tour E the original value of tour E used every edge twice this is going to work in general yes on every tree it will work because no matter what do you have when you do the tour starting from any edge you go down an edge and then eventually you come back up and you have to do it exactly once
so clearly every edge will appear twice in this E and therefore the weight is going to be twice the length of the tree twice the total weight of the tree but the weight of the tree itself is L and so the weight of E after the step two is going to be twice L
the other property about E that is important is that E must contain all the vertices in T E has to contain all the vertices in T ok so it is a tour the only problem is that it contains some vertices more than once and that’s why it cannot be its not a good tour for us
so if v appears more than once in E we remove the first appearance so this is good because we are going towards making sure that every vertex appears only once but what is this due to E in particular does it do anything bad to the weight of E ok
so here is the important claim the claim says that after step three weight of E is at most twice L it can only decrease now this is the main part of the argument and the proof is actually quite simple

so what is the new weight the new weight is the old weight ok and suppose v was the edge we deleted ok so let me take a picture to explain this

so this was a portion of E and this is the vertex V which we are deleting how do we deleted we take the previous vertex which we call u lets call it u we take the next vertex we call it d call it w and we removed these two edges and we put down this edge

so what happens to the total weight well the total weight now becomes the old weight plus what we put in or or minus what we removed so minus what we removed is so minus of D of u v and we also removed this plus D of v w and we put in u w sorry we put in ya we put in u w so this is the new weight that’s what we have written down over here (refer slide time: 33:56)

but notice that these are two sides of a triangle and this is the third side essentially ok so this is the straight path and this is the cross path so which of this two is bigger
so clearly this one is going to be bigger if at all and therefore we know that this entire thing this entire thing has to be less than or equal to zero or therefore the whole thing is at most old (refer slide time: 36:28)

so we have proved that the new weight is at most the old weight the old weight was twice L and so the new weight is also twice L

so if we keep on repeating this step as many times as we can what happens the weight keeps on reducing so it will always be bigger than it will always be smaller than twice L so we have proved that the weight of E is always going to be at most twice L so the final claim is that just before we return of course the weight is going to be at most twice L but E is also going to have every vertex exactly once why is that well we repeated until no vertex appeared more than once

so clearly no vertex appears more than once but initially every vertex did appear at least once and therefore finally every vertex appears exactly once and so therefore E is a tour every vertex appears once and its weight is twice L L is lower bound and therefore we are done

the final issue is there might be some question about the time required for this part
so here is a very nice simple observation which says that this entire thing can be done in
linear time what does this loop do so it says if v appears more than once delete the first
appearance but what if v appears several v appears several times then we will delete all
but the last appearance appearances and this is going to be true for every vertex
so we are going to keep only the last appearance of every vertex in this traversal E
but what is that we know that right when you when you do graph search we should do a
post order traversal that’s exactly what this is and therefore the time for this steps three
and four together is at most the time for a breadth for a depth first search and therefore it
is just O of the number of edges plus the number of vertices
so the total time is just simply is dominated by the time for finding the minimum
spanning tree and therefore it is E plus v log v ok say using prims algorithm
lets now consider the next problem the next problem is precedence constraint scheduling
which is also an NP complete problem and we are going to find a polynomial time
approximation algorithm for this
the input to this problem has two parts the first part is a directed acyclic graph G vertices
in this graph represent unit time tasks and there is an arc directed edge going from u to v
corresponding to the restriction that vertex u must execute before vertex v
so there is a precedence constraint from u to v and therefore the name of this problem you
are also given as a part of the input and integer p [noise] where p denotes the number of
available processors
so p is the number of tasks that you can perform at each step you may not be able to find
that many task but certainly you cannot perform more than p tasks at each step
the output the output for the output we require to specify an integer time of execution T
of u for each vertex u such that first of all T of T of u is greater than zero and at most p
vertices has the same time of execution
further more if there is an arc from u to v then the time of u is must be strictly less than
the time of v now remember that these are integers so this really means less than or equal
to so that there is a difference of at least one
and finally we want to minimize the length of the schedule so the maximum over all times is as small as possible [sneezing] this problem is known to be NP complete for variable p
so if p p changes p is allowed to change as a part of the input then this is known to be NP complete

here is one lower bound i claim that the length of the longest path in this graph is a lower bound so lets do that band lets see that and for that lets take an example as well
so lets take a simple graph so say the graph G looks something like this so here is vertex one which is one task here is vertex two which is another task then may be there is vertex three over here and there is an edge from one to three there is vertex four there is an edge from one to one to four as well may be there is an edge from two to four also may be there is a vertex five and there is an edge and say there is an vertex six with these edges

so this for example is G so this is one part of the input and lets say p is equal to two so we want to find a schedule so i claim the first lower bound which and that is that is claimed in the first lower bound that no matter what you do the length of the longest path is is lower bound at the time required (refer slide time: 43:22)

so the idea is actually fairly simple so lets identify a longest path over here
so in this case the longest path is quite simple so say for example this is the longest path there are several longest paths but this is the longest path what is this length well we are suppose to measure the length in terms of the vertices the vertex length so this has length three and the claim is that the length of the schedule must be at least three why is that well the precedence constraints says that if this is executed at step one whatever step it is executed it is executed at this can not be executed at the same step so it has to be executed one step later this has to be executed one step further than that and so on so whatever the length of the graph is that many steps are needed for this execution [sneezing] so that’s the first lower bound second lower bound is based on how much load can be consumed at each step so if n is the number of vertices in G how many time steps how many how many vertices can be consumed can be worked on by the p processors at each step well at most p and therefore n over p steps are at least needed so that lower bound in this case is six upon two which is also equal to three so the first lower bound L is equal to three the second lower bound is also equal to three so let us now consider let us now examine whether in fact the upper bound for this matches so is it match well here is one possible schedule so we will schedule this at step one we will schedule this also at step one and in fact that is our only choice next we have these three so we can pick say this we will schedule at step two this we will schedule at step two this is ready to be scheduled but we cannot schedule it because we only have two processors so this has to be schedule at step three this we have a processor available but this cannot be scheduled at step three because this has to be scheduled only after this so this has to be scheduled at step four so T of one and T of two are both ones T of three and T of four are both twos T of five is three and T of six is four so in this case the upper bound in fact is four and it is bigger than the lower bounds ok
so now i am going to describe the algorithm which will get within twice the best possible schedule and it will use these lower bounds and it will also use the notion of a ready vertex (refer slide time: 43:51)

so vertex is set to be ready or ready to be scheduled if it has no predecessors or all its predecessors have already been scheduled

so now i will describe the scheduling algorithm so this is a procedure sched which takes G and p and it’s a two approximation algorithm it produces a schedule whose length is twice the optimal schedule as we will prove in a minute

so here is the algorithm actually its quite simple while the entire graph has not been scheduled we select as many ready vertices as possible but at most p for each selected vertex u we will set T of u equal to i so we will schedule it at step i and then we will increment the time and then we will repeat

how long does this whole thing take ok well the algorithm will take time the time required will be the time to identify this ready vertices

so the ready vertices will be found by looking at by looking at vertices which have already been scheduled i will just say that this can be done very efficiently by doing a topological sort and in fact you can do the whole thing in time linear in the size of the graph (refer slide time: 46:54)
so this in fact will run in polynomial time so it is easily shown that a topological sort will suffice
let us now consider whether this is correct so is this correct well we are following the restriction about the number of processors because we are only picking at most \( p \) vertices
we are following the restriction about precedence we are because we are picking only ready vertices so this is going to produce a correct schedule a valid schedule and it is going to run in polynomial time
the only thing that we need to prove that it is a two approximation algorithm
so let let \( G_i \) denote the graph induced by the unscheduled vertices after iteration \( i \)
\( L_i \) is the length of the longest path in \( G_i \) so remember that that’s a lower bound on \( G_i \)
let \( n_i \) denote the number of vertices in \( G_i \)
the first claim is either \( n_i/p \) which is the lower bound on the \( i \)th graph is equal to \( n_i - p \) \( n_i - 1 \) upon \( p - 1 \) or \( L_i = L_{i-1} - 1 \)
so either this lower bound decreases or \( L_i \) is equal to \( L_{i-1} - 1 \) minus one minus one so either this lower bound decreases or this lower bound decreases
so after first iteration we have $L_{1}$ then we have $L_{2}$ so $L_{2}$ will be either one less or this lower bound for the second iteration will be one less and this will be enough to prove the two approximation.

so let us prove this the proof is actually quite simple so the basic step in the algorithm is to find $p$ vertices in that step three of iteration $i$.

so suppose it does find those $p$ vertices in iteration $i$ so what happens ok so if it finds $p$ vertices then the number of vertices that remain is going to be $p$ less so $n_{i}$ is going to be equal to $n_{i-1}$ upon $p$.

but now if you simply divide by $p$ then we will get part a so this happens then part a will hold.

the other case is suppose that the algorithm does not find $p$ vertices if the algorithm does not find $p$ vertices then there are at most $p$ minus one ready vertices.

(refer slide time: 48:45) so what are the ready vertices the ready vertices are the vertices in the graph such that there predecessors have already being scheduled or they don’t have any predecessors whatsoever what do we know about such vertices (refer slide time: 49:04)
well what do we know about paths here is the key idea every longest path must originate on one of these ready vertices [noise] suppose it doesn’t suppose here is a longest path well we go back this is not ready vertex so there must be vertex behind it if there is a vertex behind it then we are getting a path even longer therefore by contradiction the longest path must originate over here so the algorithm on the other hand schedules all these ready vertices but if it does schedule all these ready vertices then the lengths of all the paths starting at these ready vertices including the longest path must decrease by one but that’s essentially saying that L sub i equal to L sub i minus one minus one thus we have proved this either this holds or this holds the next claim is that this algorithm gives a two approximation so here is a proof so remember L was a lower bound the length of the longest path in the entire graph so i am going to call it L sub zero n was the number of vertices in the entire graph i am going to call it n sub zero the initial lower bounds thus are L sub zero and n sub zero upon p after iteration i the bounds are L sub i and n sub i upon p and what else to be known claim two which we just proved says that either the first bound or the second bound drops by one in each iteration
so starting from L zero and this n zero upon p we go to L one and n one upon p L two and n two upon p and so on.

Claim two says that either the first one drops or the second one drops eventually until we get to the last iteration no bound can drop below zero right because it doesn’t make sense to say that the length of the path is negative or that the number of vertices is negative. So which means that if more than L plus n over p steps are taken then one of these bounds must become negative starting from here because this L zero can only drop by L this can only drop by n over p so one of these has to go below zero but that’s not possible and therefore it means that L plus n over p steps must suffice. Our schedule must have length L plus n over p at most but what do we know about L plus n over p well this is certainly less than two times max of L and n over p (refer slide time: 51:01)

![Proof: Let \( L = L_i \), \( n = n_i \). Initial lower bounds are \( L_i \), \( n_i \). After iteration i the bounds are \( L_i \), \( n_i \). Claim 2 says that either the first bound or the second bound drops by 1 in each iteration. No lower bound can drop below 0. If more than \( L + n/p \) steps are taken, some bound becomes negative. So \( L + n/p \) steps suffice. \( L + n/p \), \( 2 \cdot \max(L, n/p) \).]

so we will just replace the smaller of the two with the max so this is going to be less than two times max of L and n over p but what is max of L L and n over p so this is a lower bound this is a lower bound so the larger of the two is also lower bound ok so but if its if its a lower bound then OPT is even bigger than this so this is less than twice OPT.
so this max is a lower bound so OPT the length of the optimal schedule cannot be smaller than the max and therefore we have that L plus n over p is less than two times OPT but this is the length of the schedule which we produced and this length is less than two times OPT so L done 
so we have proved that this algorithm gives a two approximation ok 
so now i am going to conclude so today we discussed various strategies for coping with NP complete problems the strategy which we are going to study is the strategy of devising approximation algorithms so these are defined as giving nearly good solutions rather than the best possible solutions but the good thing about them is that the time is polynomial there are various techniques for designing approximation algorithms and the techniques that we studied today can be summarized as follows (refer slide time: 53:22) 

so basically we try to find lower bounds on the objective function which we want to minimize ok and then we try to get close to this of course if the objective function had to be maximize and we will try to find upper bounds and we will try to get close to those ok so device we will algorithms in this case which will get close to this lower bounds so the lower bounds are easily should be easily identifiable and therefore we can we can actually compute them and then we can may be ta try to target an algorithm which tries to
meet them but of course it will not succeed in meeting them but it will try to it will
succeed in hopefully getting close to them
we will see more such techniques in the next lectures thank you