Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science Engineering
IIT Bombay
Lecture No - 34
Approximation Algorithms for
NP-Complete Problems – III

this is the third lecture in the series on approximation algorithms
a short review of what we have seen so far so one of the approximation algorithms we saw gave us a two factor approximation for metric TSP
we also saw a two approximation algorithm for metric clustering having seen these algorithms a natural question might be that instead of just getting a two approximation is it perhaps possible that we can get a one point five approximation say either for metric TSP or for metric clustering
here are some possible answers and these happen to be true as a matter of fact so for example for metric clustering you cannot get better approximation than a factor two unless P is equal to NP (refer slide time: 03:04)
so this is a rather interesting kind of a result that it is not only is it hard to find fast clustering algorithms which accurately cluster but even getting approximate clustering algorithms seems to be difficult because we believe that P is not equal to NP most people believe that however here is some good news if you look at the Euclidean TSP ok so which is a special case of the metric TSP it turns out that you can get any approximation factor one plus epsilon however there is a catch and the catches that your running time will also depend upon epsilon so essentially there is going to be a trade off between the running time and the approximation factor obviously the closer you want the approximation factor to one the higher is your running time going to be so the smaller the epsilon the larger the running time so this dependence will be captured somehow we will we can evaluate that dependence and in fact we can work with wide range of epsilons and that’s what this result says such results are called approximation schemes and this is what we are going to study today here is a definition an algorithm A for a problem P is said to be a polynomial time approximation scheme abbreviated as PTAS if the following conditions hold first of all A must now take two arguments of course it has to take the problem instance and it will return the problem answer but it will also take in addition a single number epsilon which is greater than zero which is going to tell the algorithm how close an answer we want how good an answer we want so A is going to return a solution with approximation ratio one plus epsilon the smaller we name this number the better is our solution going to be but as mentioned earlier the time taken by A is going to be polynomial and now its going to depend on epsilon so this polynomial we change with epsilon that’s the new addition so the time is going to be polynomial ok we are going to be able to do this for any epsilon but the time will be a function of epsilon as well
so it’s a scheme in the sense that you can operate anywhere its not just one algorithm but in fact you can think of it as a family of algorithms defined by different values of epsilon there is also a notion of fully polynomial approximation scheme which is often abbreviated as FPTAS ok and in this case we require that the time should also be polynomial in one over epsilon so yes the time will increase as you reduce epsilon but it will only increase polynomial (refer slide time: 06:42)

here however the time can increase and in fact it can increase much faster than any polynomial so this is this is going to be something harder to do or this is going to be something which is going to be faster for the same epsilon here is what we are going to today we will describe an FPTAS or a fully polynomial approximation scheme for the knapsack problem which we studied earlier the time taken by the scheme is going to be O of n cube upon epsilon so notice that this is polynomial in the size of the instance n as always we will use n to denote the instance the size of the instance its also polynomial in one over epsilon so it is n cube read it as n cube times one over epsilon so its its dependence on one over epsilon is linear so it satisfies this condition as well
so the time taken is polynomial in $n$ as well as it is polynomial in one over epsilon and therefore we will what we will get is going to be an FPTAS and so we will be designing an algorithm which will take which will be a approximation scheme
so it will take an instance as one argument and it will take the number epsilon it will return a solution with approximation ratio one plus epsilon and its time taken will be this

let me quickly remind you what the knapsack problem is we have actually we have of course studied it earlier the input consist of an array $V$ one though $n$ $V$ stands for value and $V$ of $i$ is going to be the value of the $i$th item
the second part of the argument is going to be an array $W$ one through $n$ so these are all going to be integers $V$ and $W$ are both going to be integers and $W$ of $i$ again is going to be the weight of the $n$th item
that is going to be a third argument which is $C$ again an integer and $C$ is going to denote the capacity of a knapsack
the output for the output we have to select a subset of these items one through $n$ such that the total weight is at most $C$
so think of filling this knapsack but when we fill the knapsack we should not exceed its weight capacity otherwise a knapsack will tear or something like that and we are only allowed to pick a subset of these items and furthermore we want to pick the most valuable subset (refer slide time: 09:46)
so whatever subset we pick we add up the values of the elements which we pick and that is the value that we get and we want that value to be as large as possible so it should be maximum over all possible subsets
you have of course seen this problem earlier ok we have devised an algorithm for this the algorithm was pseudo polynomial time as we discuss some time ago and in fact knapsack is NP complete
what we are going to do today is we are going to discuss a new algorithm it is again going to be based on dynamic programming
so we will have a new dynamic programming algorithm it will also be pseudo polynomial time ok
so that wont really work for work as an approximation algorithm for one thing its not an approximation algorithm its going to be an exact algorithm but the reason it wont work is that it will take pseudo polynomial time whereas we wanted to take polynomial time
so after that we will describe how we can modify the input instance to this algorithm such that we get good approximate answers but we get them fast
so that’s gong to be the interesting idea the the sort of new idea of today’s lecture is going to be this that will take a pseudo polynomial time algorithm and we will use that but instead of feeding to it the exact instance that is given to us we will feed it a different
instance perhaps sort of an approximated instance so that we will get the answers fast although we will get approximate answers [sneezing]
so let me quickly go over the dynamic programming formulation that we have studied earlier very quickly and then i will tell you however new dynamic programming formulation is going to be different
here is the old formulation so what we asked over there was what is the best value we can get for each knapsack capacity c little c where c is an integer somewhere between anywhere between one and C one and capital C
so capital C is the integer given to us as the part of the problem instance so if you remember the dynamic programming algorithm which we studied long ago it was doing exactly this
for each value of c little c between one and capital C it calculated what is the best value we can get
so basically the question was for a fixed capacity and we take different capacities but in each individual question the capacity is fixed what is the best possible value or what is the largest possible value we can get [sneezing]
the new problem or the new way of looking at this problem is to ask a different question the the question which we are going to ask is what is the lightest knapsack and by the time in what is the smallest capacity knapsack which we can use for getting some value v ok
and we will do this for all values of this little v in the range one through Vall what is Vall the Vall is the sum of the values of all the items
we are looking at this Vall because at most we will be interested in filling all items beyond that there is nothing of interest because there are no more items to fill if you figure out what is the capacity needed to fill all the items we should be we should really be happy
so this is going to be the kind of question that we are going to study today and as you can see its sort of the complementary question ok
so here we ask the largest value for fixed capacity here we are going to ask the least capacity for the fixed value or the lightest knapsack for the fixed value i want to point out
that if we answer these new questions we will still get a solution to the original problem which we started off with
so these new questions essentially compute $S$ of $V$ where $S$ of $V$ is the lightest knapsack for value $v$ that’s exactly what is being computed over here for different values of $v$
and notice that $S$ of $V$ is an increasing sequence or a non decreasing sequence
now the question that we started off with was to get the best value for capacity $C$ so on the phase of it it might seen that this is the natural question to ask and indeed it is somewhat more natural
but the point is that even if we answer these questions we will be able to find this why because we simply ask what is the largest value $v$ such that the size required for it is less than or equal to $C$ (refer slide time: 15:21)

so clearly this is going to be the value which we are going to get had we started out with a knapsack of capacity $C$ in this question
so the idea over here is that even if we solve all such questions we will be able to solve this question which we originally started off with you might think that here we are ((13:58)) to solve many many questions even though our single question over here is just there is the single question over here
but if you remember if you go back to the dynamic programming algorithm we looked at
even to answer the single question we really answered all these question so it its not that
we an we are going to answer more questions this time
well we are ok so we are not we are will we may not necessarily answer more
questions over here we are just going to answer different kind of questions and from the
from those answers as mentioned over here we will still be able to get the answer to the
the real question that we are asking [sneezing]
so from on for the rest of the lecture i am going to think of these new questions and once
we have the answers to these new questions we will use this idea to return this idea to
return the best value of capacity C for capacity C
so basically my new instances for each of these questions is are going to be characterized
by these three arguments
so we are going to have an argument W the weight the value V and the target value the
target total value that we need and we are going to have such questions for all such
different V’s
and our objective is going to be minimize the knapsack size ok so its going to be this kind
of questions where we want the the lightest knapsack for this fixed value V
so how do we solve this problem basically we you see that the algorithm is actually the
logic behind designing the algorithm is very similar to what we did for the other for the
more natural looking algorithm
basically we need to decide whether it will include item one whether it will include item
two whether it will include item three and so on so we have a series of decisions to make
so this is what we want we want the optimal solution to the instance W one through n V
one through n and v and lets look at it as this search space idea
so we consider the search space for this big instance for this instance which is W of one
through n V of one through n and little v so this is our search space
so what is contained in it it contains knapsack capacities ok or it contains solutions to
such problems ok
so it contains feasible solutions to such problems now the feasible solutions to such
problems can be of two types ok
so for example we can have a solutions which have value $V$ ok so every thing over here must have value $V$ but may be the solution contains item one or not 
so there could be one set of solutions which contain item one and another set of solutions ok which item contain item one and another set of solutions which do not contain item one 
so this consists of sets with value $V$ not containing item one and this must contain item one sets containing item one (refer slide time: 20:41) 

so we really want the lightest capacity ok set the lightest weight set from this but we set that this search space has been decomposed into two parts 
so we could ask what is the lightest capacity the lightest set in this what is the lightest set in this and we could take the lighter of the two this is precisely what has been written down over here 
so we wanted the optimal solution to the instance $W_1$ through $n$ $V_1$ through $n$ and little $v$ and we can get that by picking the lighter of the solutions or the solution with the smaller weight of these two solutions 
so the first one is we look at this set and we pick the lightest solution in that and that is over we will pick the lightest solution in that and that is over here
so if you remember this is pretty much the idea we used in our original dynamic programming algorithm as well [sneezing] ok what what can we say about these two things we can say something rather interesting
so this first term over here consist of all solutions to this instance which do not have item one
but what does that mean that just means that we might as well we asking for solutions to W two through n and V two through n and v because we are not using item one any way
so this term is in fact exactly the optimal solution to this instance so notice that this is interesting this is useful because we are heading towards a recursive solution
so a solution to this is being expressed in terms of a solution to a smaller problem that’s always good news
what about this we want the lightest solution of value v having item one so we are looking at this set we know that at at this set of sets this part of the solution space we know that every set over here contains item one
so we take that away and what is left ok so what is left are sets which do not contain item one but we also know that there value had better be this v minus the value of one why because every set in this part of the space originally had value little v if they contain an item one and if we remove that item one then new value must be exactly v minus V one
so what remains in this part of the search space are sets which do not contain one but whose value is v minus V one ok
so we take su the best amongst those and act to that the item one ok but what is the best amongst those
so the best amongst those is again an optimal solution to some instance in fact its an optimal solution to this instance
so it’s an optimal solution to W of two through n V of two through n and v minus V one
this is this is the knapsack capacity that this is the value that we are seeking
why are we seeking this smaller value because we know that we are at the end we are going to add item one to it
so if we add item one into it it wi the total value will become v and which is the value that we want
so therefore we take an optimal solution to this problem instance we take an optimal solution to this problem instance add one to it and take add the item one to it and take the lighter of these two things [sneezing] so that’s basically the algorithm ok
so now all that we need to do is express this as a recurrence ok well there is a slight catch so when we do this this v minus V one could become negative what is that mean i want an optimal solution in which the value is negative that doesn’t mean anything right so we had better be better be careful about that so we must generate this part only if v minus V one is greater than or equal to zero because otherwise this problem instance is undefined
so when we write down our occurrence we will have to put an explicit check whether v minus V one is greater than or equal to zero or whatever greater than or less than zero ok ok (refer slide time: 28:12)

so here is our expression in terms of which we are going to define our recurrence so S i v i am gong to define as the least capacity knapsack which can give the value v using items i through n
so this i defines this i this is the same i and this v defines the value that we want
so \( S_i v \) is going to denote the least capacity knapsack which gives me value \( v \) using items i through n

so now i am just going to take this expression that we derived and express it in terms of terms of this kind

so what is this optimal solution to \( W V v \) well this is simply this left hand side is simply \( S \) one \( v \) because we are starting with \( i \) equal to one we are allowing all items and we are asking for value \( v \)

then we are going to take the lighter of the solutions so correspondingly we are going to use the minimum over here ok the minimum of two solutions what is the first solution the first solution is to the instance \( W \) two \( n \) \( V \) two \( n \) little \( v \)

so its going to be the least capacity solution to this instance so that’s as good as saying it is \( S \) two \( v \) so that’s what we have written down over here

the second part is this however when we write this down we have to make sure that \( v \) minus \( V \) one is not less than zero so let us check that

so i am going to write down a c style expression ok so this says that let us check whether \( v \) minus \( V \) one is greater than or equal to zero

if it is then the value that we want here is \( W \) of one plus \( S \) of two \( v \) minus \( V \) one why is that because we are going to have item one always and therefore we are going to have the weight corresponding to item one always present over here

and we are going to add this item one into the optimal solution to this problem but the weight of that optimal solution ok the capacity needed for that optimal solution is simply \( S \) of two we start with items two through n and therefore this is a two over here and the and the value that we are expecting is \( v \) minus \( V \) one exactly this

so if this expression is greater than or equal to zero then this is the value that we want if this expression is less than zero this problem is undefined but what does how do we represent that ok

so that we represent by putting in an infinity so we are taking the min this infinity will never be taken and if if this second expression is infinite then that’s as good as saying just give me the first expression ok
so this is what we have distilled out of this well its actually the same thing but its now return out more compactly in terms of variables of this kind subscripted expressions of this kind [sneezing]
you must of course generalize it in in order to use it to design an algorithm we need to generalize it ok
so here is a generalization so here we were looking at all the items one through n and you can note you can see that internally we got we needed to have solutions to problems in which we were having items two through n
if we did recursion on that then we would get three through to n four to n and so on so therefore we now consider the more general case
so we are going to ask what is the least capacity solution of value v using items i through n so that is S of i v that is denoted by S of i v
so analogously we will write down the expression the more general expression so here we skip the first item and so we started of with two so similarly here we are going to sta start of with i plus one ok so that’s the that’s the first solution re corresponding to this ok
and instead of checking whether v minus V one is greater than or equal to zero we are simply going to be checking whether v minus V of i is greater than or equal to zero if so here we took W one plus something here we will take W i plus something again this something is going to contain items two through n over here it will contain items i plus one through n over here and so we will have i plus one over here
this was the value required was v minus V one here the value require is v minus V i because we are adding in item i later on anyway in case we want v minus V i that is the value of this part and then to to it we add item one
so the weight solution value of this the whole thing together will be W of i plus S of i plus one v of V minus i as before
and of course if v minus V i is less than zero then we don’t want this entire term to be taken into account at all and therefore otherwise we are going to put down an infinity ok so this is a defining recurrence that we are going to use so let us see how this recurrence can be solved ok
so as usual we are going to be keeping a table ok so i have just written out that recurrence again so let us see what kind of table we can use for this
so here is the table this is the i axis going down vertically this is the v axis ok
so earlier i said that v really needs to start from one but its useful to have a zero here as well ok
so have started it of from zero and as before it goes to Vall so since we are going to have since i want to show you what is recurrence means in this table i have put down some specific en i have marked out from specific entries
so lets ask how this rec recurrence will work out in this table so i want to compute S of i v which is this entry in this table ith column vth row i am sorry ith row vth column
now this entry depends upon which entries well it depends upon this entry and it depends upon this entry doesn’t we just don’t have to look at this entry we have to do something to it but this other thing that we have to do we know we know what the value of W i is so it really depend upon this entry and this entry ok so which are these two entries so this entry is this entry and this entry is this entry ok
of course if v minus V i was less than zero then this entry would would fall outside the table and so we would only have a single entry over here but this is the more common more interesting case
so to fill this entry its it is sufficient if we have this entry and this entry filled that’s all coming out of this recurrence which we have written down over here well that suggests a way of filling in this table
so we sort of fill in going bottom up or bottom right from we start of at the bottom and go upwards but we also need this and so therefore we also have to start from the left side so to do that we would need to have these entries filled how do we fill this entries well lets interpret what these entries are (refer slide time: 34:38)
so this entry in general is going to be $S \, i \, 0$ for different values of $i$ and let me remind you what $S \, i \, 0$ is

$S \, i \, 0$ is the capacity the minimum capacity needed to get a value of zero using items $i$ through $n$ that doesn’t seem to difficult does it we just want to get a value of zero

so what is the minimum capacity [noise] we need well trivially the answer is zero capacity ok so if we get capacity if we get a knapsack of capacity zero ok we can certainly get value of zero by filling nothing into it and clearly there is no smaller knapsack that we can use because this is the smallest possible knapsack [sneezing]

so this entire yellow column just needs to be filled with zero’s and that something we can do without any computation

so that leaves open the question of how do we well this row

so lets try to interpret what this row is so this row in general is going to be $S \, n \, v$ where which is denoted which is denoting the capacity needed to get value $v$ the least capacity needed to get value $v$ using item $n$ alone

so you are just allowed to pick item $n$ nothing else so what kind of capacities can you get what kind of values can you get well clearly if this $v$ happens to be $V$ of $n$ then you can do that using just a single item ok
so that would been that if \( v \) happens to be \( V \) of \( n \) then the least capacity knapsack that you would need would have to have capacity of \( W \) of \( n \) so one of these entries can be filled using this what about the rest of the entries suppose we want to do this for a different value of \( v \) so lets say we want to get a value larger than \( v \) of \( n \) can we do it we are only use to we are only allowed to use the item \( n \) so clearly we cannot do it we cannot get a value either bigger or larger than \( V \) of \( n \) by using only item \( n \) so then what we do so that can be represented quite nicely by saying that by putting in an \( \infty \) infinity in this in these entries so \( v \) of \( V \) \( n \) so if \( v \) is not equal to \( V \) \( n \) then we cannot get that value and so we will say that a capacity knapsack of capacity infinite is needed let me explain why this works basically later on we are going to take things like min of this or min of this or something like that so if there are infinities over here then that value is essentially going to be ignored or if both of these values are infinities and if you take min of those infinities than an infinity will crop up over here but says that even this value is impossible to accomplish so that’s that’s sort of a nice thing that’s kind of a nice coding that infinity allows us to accomplish [sneezing] so basically now we have express the algorithm entirely so we have entries to fill and thee are sort of three kinds of entries to fill we have these yellow entries to fill we have these blue entries to fill and then there are the rest of the entries which we fill according to this recurrence so we can write done our algorithm so i am going to call my algorithm \( k \) as knapsack its going to take as argument ok so that brings us to our algorithm we will call this \( KS \) for knapsack it takes as arguments the value the weight of all the \( n \) item and the capacity so we are going to solve we are going to find an answer to this but we are going to find an answer using our new formulation so as mentioned earlier ok so there were those yellow entries to be filled they were all going to be filled with zero so we will do that ok so these entries using this equation
then there were the blue entries to be filled ok the blue entry is where that for if \( v \) is equal to \( V \) of \( n \) then the knapsack capacity needed was \( W \) of \( n \) otherwise the knapsack capacity needed was infinity
so this is how you fill the bottom row so this is how we filled and this was the equation used ok
finally the rest of the entries were filled using the recurrence which we derived so this is that recurrence (refer slide time: 38:32)

at the end of it to get the the value for the capacity \( C \) we needed to find as we described earlier the largest \( v \) such that \( S(v, 1) \) is less than \( C \)
so what is \( S(v, 1) \) so we look at all possible \( v \)'s so this is the this is column one and we know that as we go as we go down this is going to be non decreasing so we can easily find the largest \( v \) such that \( S(v, 1) \) is less than \( C \) ok and that is the value of whi which we will call \( v^* \) [sneezing]
so that is the value that we are going to get if we in fact had a capacity of \( C \) so if you are just interested in the value then we would be done at this point
however if we wanted the items to be returned as well then we can do that also you remember how we did that for the older algorithm we just had to keep track of some additional pointer some additional data structures basically we can do that as well so corresponding to every entry of the table we can determine what the corresponding set is going to be so we can also return the set of items so how long does it take just to compute everything from here until v star well these are all this loop will take time O of n this will take time O of Vall and here we have nested loops and therefore it will take time O of n times Vall in fact if you compute the set itself we will have to keep some additional data structures [noise] but as explained in the previous algorithm we can use exactly the same ideas to do the to compute the set as well in exactly the same amount of time well to within constant factors so in O of n times Vall time we can not only compute v star but we can also compute the corresponding set so this entire problem can be solved n time O of n times Vall ok so that finishes the first task that we undertook so we now have a dynamic programming algorithm which finishes in time n times Vall so now we come to the approximate algorithm here is the important point so here we are going to be allowed an error in the answer we are only required to get within one plus epsilon so epsilon is sort of the error we are allowed the point is that if we are allowed an error then it means that we can calculate using low precision so basically we are going to calculate using low precision and that is going to allow is to reduce the time so we are going to write a new new procedure a new algorithm which we which we are going to call approximate case approximate knapsack or AKS its going to take the same arguments as before but its also going take an argument called delta which is somehow going to reflect the precision and we will tight up to this epsilon later on so delta and epsilon will be related how they will be related we will specify pretty soon here is the algorithm
so we are going to define a new array $V'$ so $V'$ is simply going to be $V$ divided by $\delta$ corresponding elements ok
so every element here is going to be divided by $\delta$ and we get the corresponding element of here but we want $V$ to be integer as well
so if we get a fraction which we will in general do we will take the floor we will take the largest integer less than ok or so the or the floor
and then so essentially we are scaling down the $V$ values by a factor $\delta$ we will call KS using those scale down values but the scale down values are not really going to be important for the final answer
so for the final answer we want to return $\delta$ times whatever scale down values we got we got
so this is expected to do roughly the same job right we scale the values down we got a good solution and then we return we scale the values up the only catch is that here we took the floor so this will produce some error ok
at a same time sense the answer the time taken for this case is proportional to the second argument the time over here is going to be $V'$ prime rather than $V$ so that’s where we are going to say on time as well
so we will say on time at a cost of some error so what remains now is to analyze all these lets say that $S$ denotes the set returned by our original KS call so where we were using the actual values of $V$ as given (refer slide time: 45:31)
so this is truly the optimal set which was return and its value x is the actual optimal value and i can think of this as the full precision problem or the full precision answer as well S prime is the set returned for this problem ok this by this case without multiplication by delta for the minute ok

lets say its value is X prime and this is a low precision answer so what we now need to do to relate this X prime that we ge so X prime and X and so on so lets just do that so i observe first that X prime is the value of the sets over here value of the set over here and what is that ok so it’s the values of all the elements in the set so V prime of i because this time we had passed V prime where i belongs to this S now this S prime [sneezing]

now here is the important point this X prime this value is bigger than this expression as well ok notice that the only difference between these two things is that instead of choosing S prime i am choosing S so this was the optimal set which was returned this is not the optimal set but its some difference set

so what happens if it’s a difference set so its value need not be as big as the value of this so in other words this value could be has to be at least as big as this value but notice that this S was an acceptable solution to this and in going from here to here we have not changed the weights so so this set is also an acceptable solution to this
so therefore we can clearly say that this value had better be no smaller than this value
because this is also feasible solution to the V prime problem as well ok
so S is also a feasible solution and therefore we get this now finally we observe that if i
take the floor of any number it is bigger than that number minus one and therefore this is
the floor of this and therefore this is bigger than this minus one ok
so V prime of i is the floor of this and therefore V prime of i is bigger than V of i upon
delta minus one because of this
so we have been able to relate X prime to this quantity over here ok so the point is that
we are getting towards the optimal set somehow we want to relate it to the value of the
optimal solution
so lets now ask what is the value that AKS will return AKS returns delta times this value
correct delta times this value this is what we do over here ok and what is that so delta
times that value well for X prime i am going to substitute this so i am going to get this
multiplied by delta
so if i multiply by delta this delta is going to get cancelled out so i get a V mi V minus i
and instead of this one i am going to get a delta because i multiplied by delta
so that explains this part now what is summation over S of Vi that simply X ok so i get an
X over here and there was a minus delta but it was also in the sum so my delta is a
constant delta does not vary depending upon which mem element of the set i am
considering ok
so i will get minus delta as many times as the cardinality of S so i am going to get X
minus cardinality of S times delta
so here is an important observation we have proved that the value returned approximately
is at least the actual value minus cardinality of S times delta
so now we are ready to evaluate the approximation ration what is the approximation ratio
so its ((46:08)) the value of the returned value upon the value of the value returned by the
approximate solution remember that this problem is a maximization problem
therefore the optimal solution has the largest possible value and in that case we define our
approximation ratio as the optimal solution upon the approximate solution so that’s
exactly what we are doing over here
so we want the approximation ratio of $X$ upon $Y$ ok so let us collect the equal the equality inequalities which we had over there so we had one inequality which is $Y$ is greater than or equal to $X$ minus cardinality $S$ times delta ok so this is one inequality (refer slide time: 47:27)

and i can write this as $X$ is less than or equal to $Y$ plus cardinality of $S$ times delta if i divide the whole thing by $Y$ i am going to get $x$ upon $Y$ is less than or equal to one plus cardinality of $S$ times delta upon $Y$ this is what we are going to get so this is this is what we finally have over here

what is the time taken by AKS ok so its $n$ times Vall $V$ prime all ok or $V$ prime all is essentially Vall by delta so it is this time ok
so notice that the time over the exact evaluation has reduced by a factor delta so all that remains now is to choose delta carefully

now here is the clever choice we are going to choose delta equal to epsilon times Vall upon $n$ square so lets just do that
so we choose delta equal to epsilon times Vall upon $n$ squared so what does that do for us so first of all this was our approximation ratio (refer slide time: 48:20)
so if we substitute into that what do we get so we substitute delta upon Y over here so we
get epsilon Vall and then this V of the Y remains as it is from here and we get an n
squared over here
so this is the approximation ratio so what becomes to this what becomes of this i claim
that this becomes at most one plus epsilon why so lets see that
the first observation is that cardinality of S which appears over here has to be less than n
after all what is S it’s a subset of n element so its cardinality has of course to be less than
n ok so that’s one important observation
then second Vall is the sum of all the elements so clearly it has to be less than Vmax
times n so if Vmax denotes the maximum value if i multiplied by n i certainly should get
something which is bigger than just the mere some of values so this is what i get and i
claim that Vmax cannot be bigger than Y ok
so the maximum item had better be accommodatable in my knapsack otherwise i would
not have considered it in my list in the first place and therefore the optimal solution had
better include this largest item and therefore Vmax times n has to be at most Y times n
so it follows from this that S times Vall is less than Y times n squared so this S times this
Vall is less than Y times n squared and therefore our approximation ratio is one plus epsilon
i also claim that the time taken is $n^3$ upon epsilon and this is much easier to see so
time taken is $n$ times $V_{all}$ upon delta so i just substitute and i get instead of delta i
substitute this so this $V_{all}$ cancels and then i get an $n$ squared here so i get $n^3$ upon
epsilon
so what has happened we have shown that our approximation ratio is one plus
epsilon so no matter what epsilon you give me i will be able to get this time if i choose
delta equal to this in my procedure and furthermore my time taken is going to be $n^3$
upon epsilon so which is exactly what we have promised (refer slide time: 50:50)

we had promised to get an approximation scheme in which the approximation ratio is one
plus epsilon for every epsilon and that we would prove that the time taken is polynomial
in $n$ the input instance length and one over epsilon that is what we have done
so that concludes the main part of the lecture i just want to make a few remarks our FPTS
FPTAS had time of $n^3$ upon epsilon it is possible to device another FPT FPTAS with
a different expression $n \log$ one over epsilon plus one over epsilon to the four
so its very likely that in practice this approximation in in many practical situation this
might be a better algorithm than this
but but this this is of course more complicated as well and we are not going to look at it
now here is an interesting fact that such as that somehow that knapsack problem is actually among the easier NP complete problems in the following sense knapsack problems in which weights and values are drawn uniformly at random from the interval zero through one [sneezing] can be shown to take polynomial them on the average so in some sense the FPTAS result says that its easier to approximate and this says that its also easier on the average in fact its polynomial in the average finally i just want to say that many problems having pseudo polynomial time algorithms even if there NP complete can be shown to have a PTAS or even an FPTAS (refer slide time: 52:30)

one example is the sub another example is the subset problem thank you