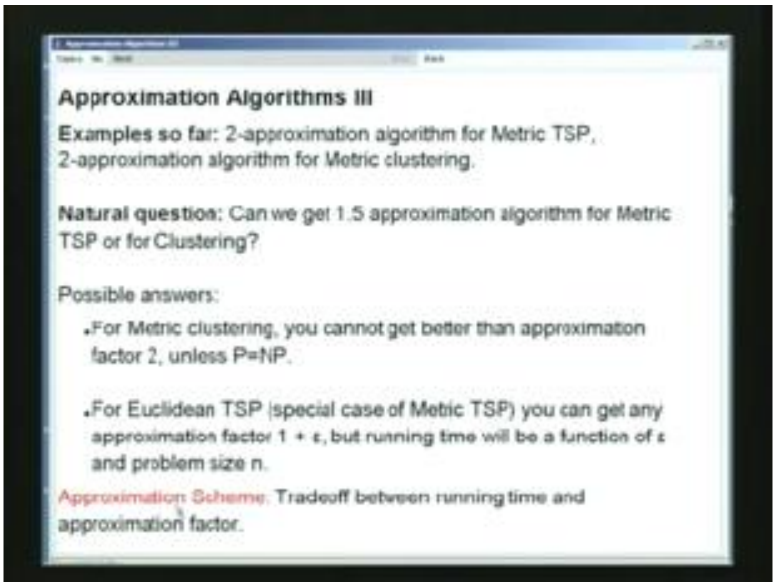
Design & Analysis of Algorithms

Prof. Abhiram Ranade
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 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.

(Refer Slide Time: 00:55)



Approximation Algorithms III

Examples so far: 2-approximation algorithm for Metric TSP, 2-approximation algorithm for Metric clustering.

Natural question: Can we get 1.5 approximation algorithm for Metric TSP or for Clustering?

Possible answers:

- For Metric clustering, you cannot get better than approximation factor 2, unless P=NP.
- For Euclidean TSP (special case of Metric TSP) you can get any approximation factor 1 + ε, but running time will be a function of ε and problem size n.

Approximation Scheme: Tradeoff between running time and approximation factor.

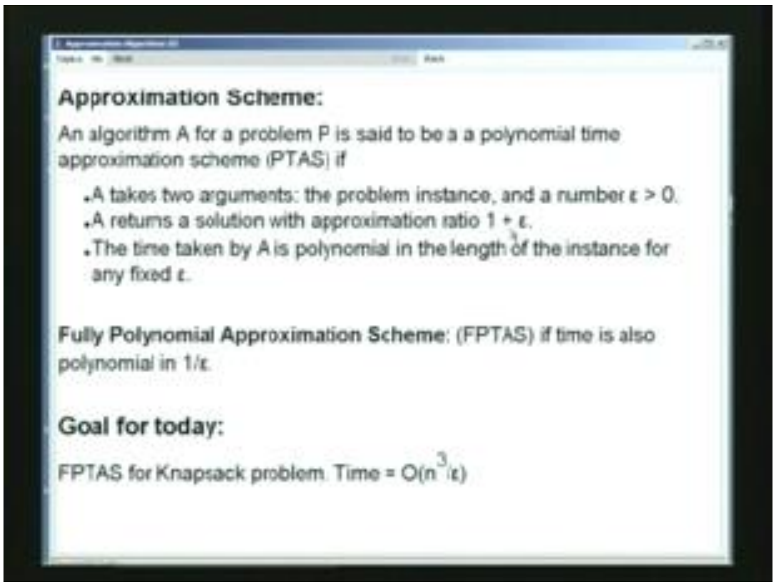
So, one of the approximation algorithms we saw gave us, a two factor approximation for Metric TSP. We also saw a two approximation algorithm for Metric clustering. Having seen these algorithms, a natural question might be that instead of just getting a two approximation is it perhaps possible that, we can get a 1.5 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. So, this is the other 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, so which is a special case of the metric TSP. It turns out that, you can get any approximation factor 1 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 of 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 can evaluate that dependence. And in fact, we can work with a wide range of epsilons. And that is, what this result says. Such results are called approximation schemes. And this is, what we are going to study today.

(Refer Slide Time: 03:13)



Approximation Scheme:

An algorithm A for a problem P is said to be a a polynomial time approximation scheme (PTAS) if

- A takes two arguments: the problem instance, and a number $\varepsilon > 0$.
- A returns a solution with approximation ratio $1 + \varepsilon$.
- The time taken by A is polynomial in the length of the instance for any fixed ε.

Fully Polynomial Approximation Scheme: (FPTAS) if time is also polynomial in $1/\varepsilon$.

Goal for today:

FPTAS for Knapsack problem. Time = $O(n^3/\varepsilon)$

Here is a quick 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 0 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 one1 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, it is going to depend on epsilon. So, this polynomial we change with epsilon that is the new addition. So, the time is going to be polynomial. We are going to be able to do this for any epsilon but, the time will be a function of epsilon as well.

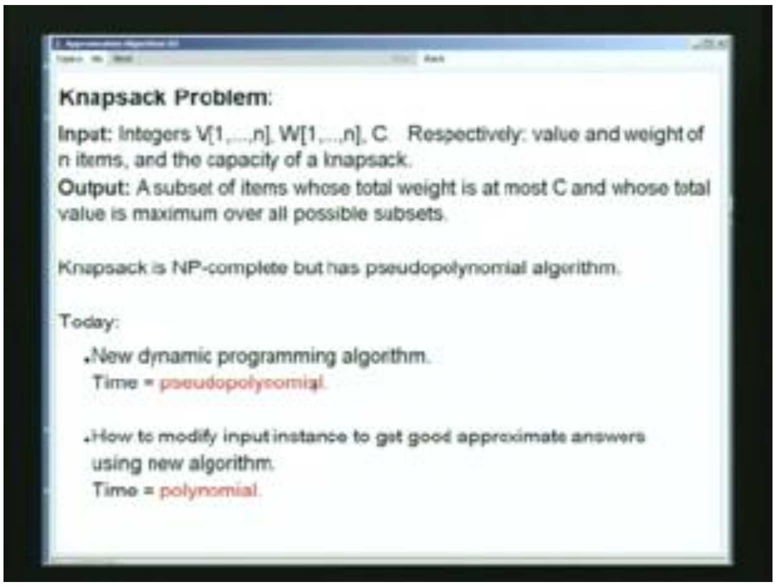
So, it is a scheme in the sense that, you can operate anywhere. It is not just one algorithm but in fact, you can think of it as a family of algorithms, defined by different values of epsilon. There is also a notion of fully polynomial approximation scheme, which is often abbreviated as FPTAS. And in this case, we require that the time should also be polynomial in 1 over epsilon. So, yes the time will increase as you reduce epsilon. But, it will only increase polynomial.

Here however, the time can increase and in fact, it can increase much faster than any polynomial. So, 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. It is also polynomial in 1 over epsilon. So, it is n cube. Read it as n cube times 1 over epsilon. So, it is dependence on 1 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 1 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 an 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 1 plus epsilon. And its time taken will be this.

(Refer Slide Time: 06:49)



Knapsack Problem:

Input: Integers V[1,...,n], W[1,...,n], C. Respectively: value and weight of n items, and the capacity of a knapsack.

Output: A subset of items whose total weight is at most C and whose total value is maximum over all possible subsets.

Knapsack is NP-complete but has pseudopolynomial algorithm.

Today:

- New dynamic programming algorithm.
  Time = pseudopolynomial.
- How to modify input instance to get good approximate answers using new algorithm.
  Time = polynomial.

Let me quickly remind you, what the knapsack problem is. We have actually, we have of course studied it earlier. The input consists of an array V 1 though n, V stands for value and V of i is going to be the value of the ith item. The second part of the argument is going to be an array W 1 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 nth 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 1 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.

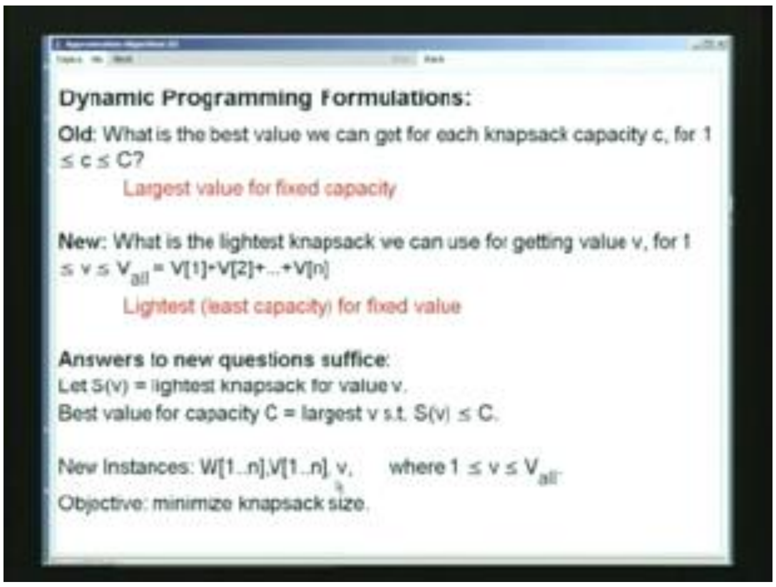
And we are only allowed to pick a subset of these items. And furthermore, we want to pick the most valuable subset. So, whatever a subset we pick, we add up the values of the elements which we pick and that is the value that we get. And we want that value to be as large as possible. So, it should be maximum over all possible subsets. You have of course, seen this problem earlier. 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. So, that would not really work, for work as an approximation algorithm.

For one thing, it is not an approximation algorithm. It is going to be an exact algorithm. But, the reason it would not 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 is going to be the interesting idea.

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.

(Refer Slide Time: 10:05)



Dynamic Programming Formulations:

Old: What is the best value we can get for each knapsack capacity c, for $1 \leq c \leq C$?

Largest value for fixed capacity

New: What is the lightest knapsack we can use for getting value v, for $1 \leq v \leq V_{all} = V[1]+V[2]+...+V[n]$

Lightest (least capacity) for fixed value

Answers to new questions suffice:

Let S(v) = lightest knapsack for value v.

Best value for capacity C = largest v s.t. $S(v) \leq C$.

New Instances: W[1..n],V[1..n], v, where $1 \leq v \leq V_{all}$.

Objective: minimize knapsack size.

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 1 and C 1 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 1 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? The new problem or the new way of looking at this problem, is to ask a different question. 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. And we will do this for all values of this little v, in the range 1 through V all.

What is V all? The V all is the sum of the values of all the items. We are looking at this V all, 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, it is a sort of the complementary question. 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 is 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.

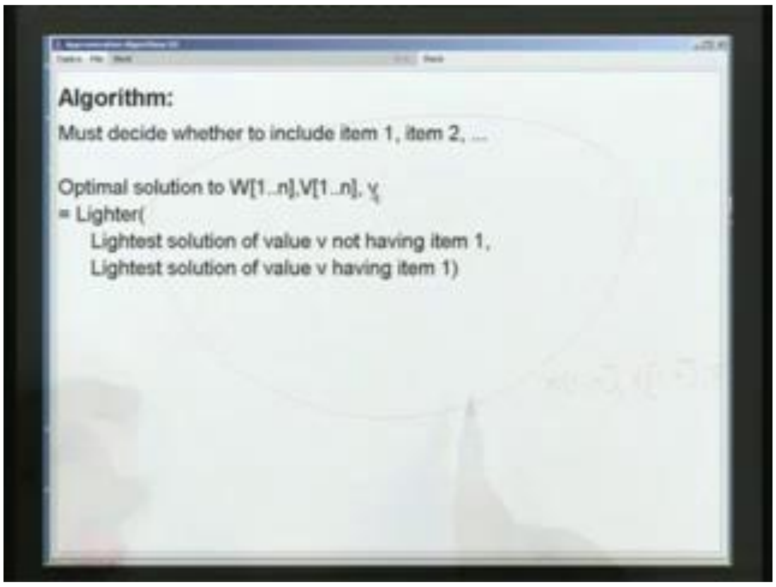
Because, we simply ask what is the largest value v such that, the size required for it is less than or equal to C. So, clearly this is going to be the value which we are going to get, had we started out with a knapsack of capacity C in this question. So, the idea over here is that, even if we solve all such questions we will be able to solve this question, which we originally started off with.

You might think that, here we are mobilized 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 this question. So, it is not that we are going to answer more questions, this time. Well, we are.

So, we may not necessarily answer more questions over here, we are just going to answer different kind of questions. And from those answers as mentioned over here, we will still be able to get the answer to the real question that, we are asking. 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 the best value of capacity C, for capacity C.

So, basically my new instances for each of these questions 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. So, it is going to be, this kind of questions where we want the lightest knapsack for this fixed value V.

(Refer Slide Time: 15:26)



Algorithm:

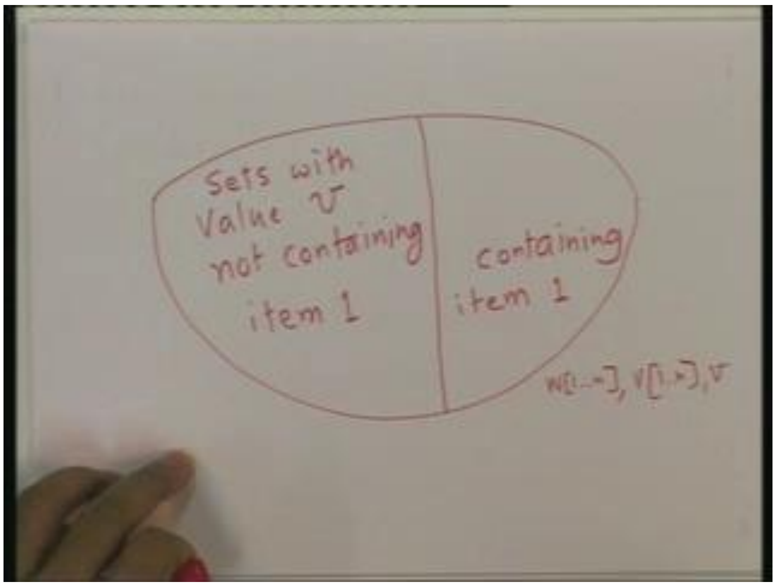
Must decide whether to include item 1, item 2, ...

Optimal solution to W[1..n],V[1..n], v
= Lighter(
    Lightest solution of value v not having item 1,
    Lightest solution of value v having item 1)

So, how do we solve this problem? Basically, we 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 1, whether it will include item 2, 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 1 through n V 1 through n and v. And let us 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 1 through n V of 1 through n and little v. So, this is our search space. So, what is contained in it?

(Refer Slide Time: 16:25)

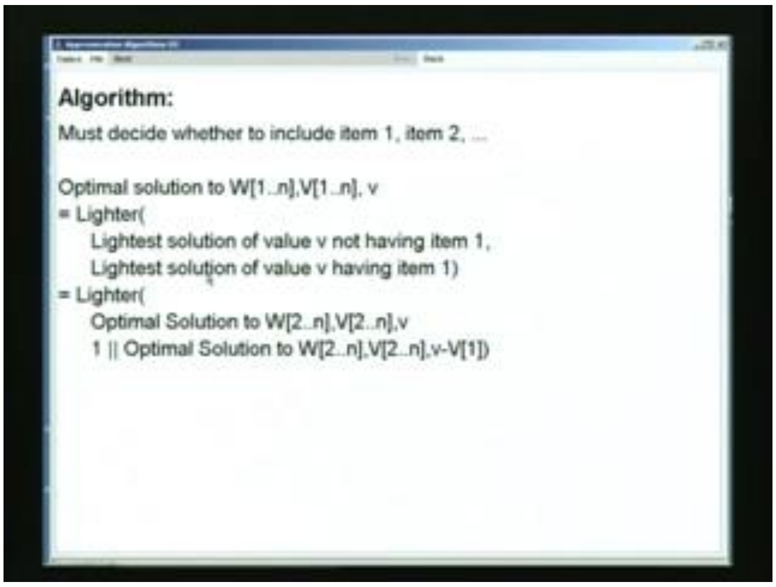


It contains, knapsack capacities or it contains solutions to such problems. So, it contains feasible solutions to such problems. Now, the feasible solutions to such problems can be of two types. So, for example we can have solutions which have value V. So, everything over here must have value V. But, may be the solution contains item 1 or not. So, there could be one set of solutions, which contain item 1 and another set of solutions which do not contain item 1.

So, this consists of sets with value V, not containing item 1. And this must contain item 1, sets containing item 1. So, we really want the lightest capacity 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 here. We will pick the lightest solution in that and that is over here. So, if you remember this is pretty much idea we used, in our original dynamic programming algorithm as well. What can we say about these two things? We can say something rather interesting.

(Refer Slide Time: 18:54)



Algorithm:

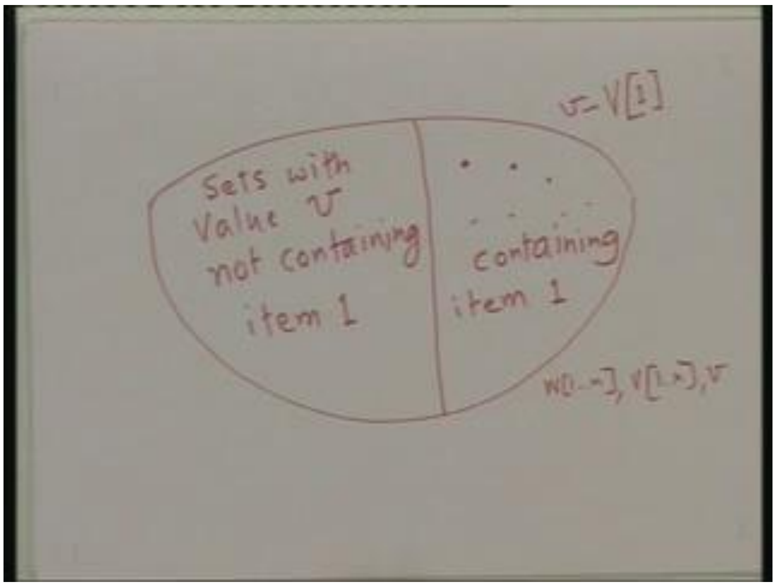
Must decide whether to include item 1, item 2, ...

```
Optimal solution to W[1..n],V[1..n], v
= Lighter(
    Lightest solution of value v not having item 1,
    Lightest solution of value v having item 1)
= Lighter(
    Optimal Solution to W[2..n],V[2..n],v
    1 || Optimal Solution to W[2..n],V[2..n],v-V[1])
```

So, this first term over here consist of all solutions to this instance, which do not have item 1. But, what does that mean. That just means, that we might as well we asking for solutions to W 2 through n and V 2 through n and v, because we are not using item 1 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.

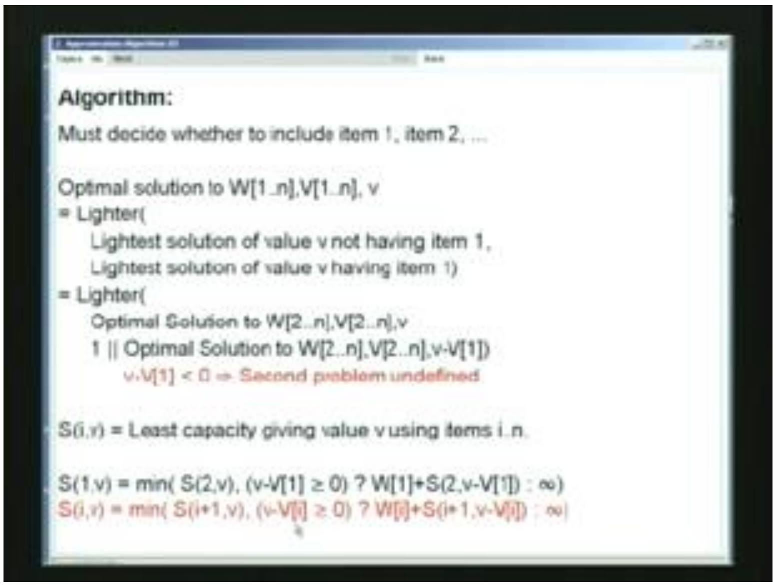
So, a solution to this is being expressed in terms of a solution to a smaller problem. That is always good news. What about this? We want the lightest solution of value v, having item 1. So, we are looking at this set. We know that at this set of sets, this part of the solution space.

(Refer Slide Time: 19:57)



We know that, every set over here contains item 1. So, we take that away. And what is left. So, what is left are sets which do not contain item 1. But, we also know that there value had better be this, v minus the value of 1. Why? Because, every set in this part of the space, originally had value little v. If they contain an item 1 and if we remove that item 1, then the new value must be exactly v minus V 1. So, what remains in this part of the search space are sets, which do not contain 1. But, whose value is v minus V 1.

(Refer Slide Time: 20:46)



Algorithm:

Must decide whether to include item 1, item 2, ...

```
Optimal solution to W[1..n],V[1..n], v
= Lighter(
    Lightest solution of value v not having item 1,
    Lightest solution of value v having item 1)
= Lighter(
    Optimal Solution to W[2..n],V[2..n],v
    1 || Optimal Solution to W[2..n],V[2..n],v-V[1])
        v-V[1] < 0 ⇒ Second problem undefined
```

S(i,v) = Least capacity giving value v using items i..n.

$S(1,v) = \min(S(2,v), (v-V[1] \geq 0)\ ?\ W[1]+S(2,v-V[1]) : \infty)$
$S(i,v) = \min(S(i+1,v), (v-V[i] \geq 0)\ ?\ W[i]+S(i+1,v-V[i]) : \infty)$

So, we take the best amongst those and act to that, the item1. But, what is the best amongst those. So, the best amongst those is again an optimal solution to some instance. In fact, it is an optimal solution to this instance. So, it is an optimal solution to W of 2 through n, V of 2 through n and v minus V 1. This is the knapsack capacity that, this is the value that we are seeking. Why we seeking this smaller value? Because, we know that we are, at the end we are going to add item 1 to it.

So, if we add item 1 into it, 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 1 to it and take. Add the item 1 to it and take the lighter of these two things. So, that is basically the algorithm. So, now all that we need to do is, express this as a recurrence. Well, there is a slight catch.

So, when we do this, this v minus V 1 could become negative. What is that mean. I want an optimal solution in which the value is negative, that does not mean anything. So, we had better be careful about that. So, we must generate this part. Only if v minus V 1 is greater than or equal to 0, 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 1 is greater than or equal to 0 or whatever, greater than or less than 0. So, here is our expression in terms of which we are going to define our recurrence. So, S i v I am going 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 this kind. So, what is this optimal solution to W V v. Well, this is simply this left hand side is simply S 1 v. Because, we are starting with i equal to 1, 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, the minimum of two solutions.

What is the first solution? The first solution is to the instance W 2 n V 2 n little v. So, it is going to be the least capacity solution to this instance. So, that is as good as saying, it is S 2 v. So, that is 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 1 is not less than 0. So, let us check that. So, I am going to write down a c style expression.

So, this says that, let us check whether v minus V 1 is greater than or equal to 0. If it is, then the value that we want here is W of 1 plus S of 2 v minus V 1. Why is that? Because, we are going to have item 1 always and therefore, we are going to have the weight corresponding to item, always present over here. And we are going to add this item 1, into the optimal solution to this problem.

But, the weight of that optimal solution, the capacity needed for that optimal solution is simply S of 2. We start with items 2 through n. And therefore, this is a 2 over here. And the value that we are expecting is v minus V 1, exactly this. So, if this expression is greater than or equal to 0, then this is the value that we want. If this expression is less than 0, this problem is undefined. But, what does. How do we represent that? So, that we represent by putting in an infinity.

So, we are taking the min, this infinity will never be taken. And if this second expression is in infinite, then that is as good as saying just give me the first expression. So, this is what we have distilled out of this. Well, it is actually the same thing but, it is now return out more compactly in terms of variables of this kind, subscripted expressions of this kind. You must of course, generalize it in order to use it to design an algorithm, we need to generalize it.

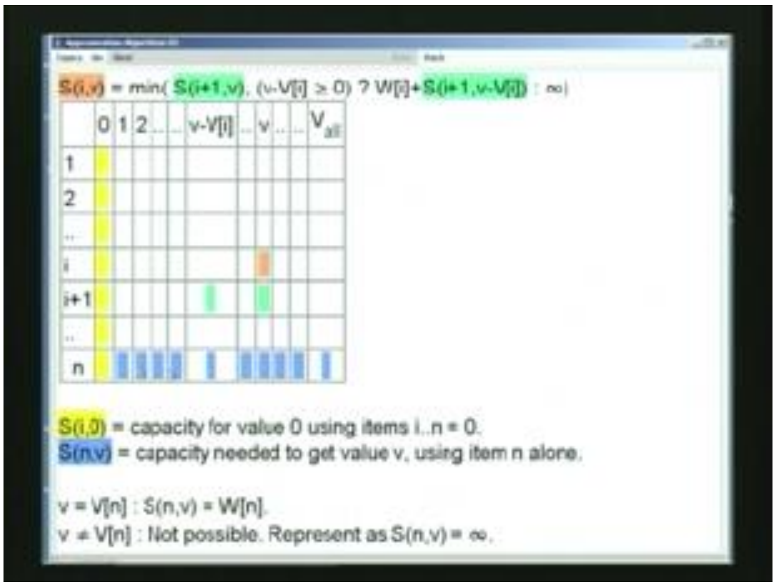
So, here is a generalization. So, here we were looking at all the items 1 through n. And you can note, you can see that internally we got we needed to have solutions to problems in which, we were having items 2 through n. If we did recursion on that, then we would get 3 through to n 4 to n and so on. 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 off with 2. So, similarly here we are going to start off with i plus 1. So, that is the first solution corresponding to this. And instead of checking, whether v minus V 1 is greater than or equal to 0. We are simply going to be checking, whether v minus V of i is greater than or equal to 0.

If so here we took W 1 plus something. Here we will take W i plus something. Again, this something is going to contain items 2 through n over here. It will contain items i plus 1 through n over here. And so we will have i plus 1 over here. This was the value required, was v minus V 1. Here the value require is v minus V i, because we are adding an item i later on anyway. In case we want v minus V i that is the value of this part and then to it, we add item 1.

So, the weight solution value of this whole thing together will be W of i plus S of i plus 1 v of V minus i, as before. And of course, if v minus V i is less than 0, then we do not want this entire term to be taken into account at all. And therefore, otherwise we are going to put down infinity. So, this is a defining recurrence that we are going to use. So, let us see how this recurrence can be solved.

(Refer Slide Time: 28:41)



S(i,v) = min(S(i+1,v), (v-V[i] ≥ 0) ? W[i]+S(i+1,v-V[i]) : ∞)

	0	1	2...	...	v-V[i]	...	v	...	...	V_{all}
1	🟨									
2	🟨									
..	🟨									
i	🟨						🟧			
i+1	🟨				🟩		🟩			
..	🟨									
n	🟨	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦

S(i,0) = capacity for value 0 using items i..n = 0.
S(n,v) = capacity needed to get value v, using item n alone.

v = V[n] : S(n,v) = W[n].
v ≠ V[n] : Not possible. Represent as S(n,v) = ∞.

So, as usual we are going to be keeping a table. 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. So, earlier I said that v really needs to start from 1. But, it is useful to have a 0 here, as well. So, have started it off from 0 and as before it goes to V all. 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, I have marked out from specific entries. So, let us ask how this 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, ith row vth column. Now, this entry depends upon which entries. Well, it depends upon this entry. And it depends upon this entry. It does not, we just do not have to look at this entry.

We have to do something to it. But, this other thing that we have to do, we know what the value of W i is. So, it really depends upon this entry and this entry. So, which are these two entries? So, this entry is this entry and this entry is this entry. Of course, if v minus V i was less than 0, then this entry 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, it is sufficient if we have this entry and this entry filled. That is, 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, let us interpret what these entries are. 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 0 using items i through n. That does not seem to difficult does it, we just want to get a value of 0. So, what is the minimum capacity, we need. Well trivially, the answer is 0 capacity.

So, if we get capacity, if we get a knapsack of capacity 0 we can certainly get value of 0 by filling nothing into it. And clearly there is no smaller knapsack that, we can use, because this is the smallest possible knapsack. So, this entire yellow column just needs to be filled with 0s. And that something, we can do without any computation. So, that leaves open the question of, how do we fill this row. So, let us try to interpret what this row is.

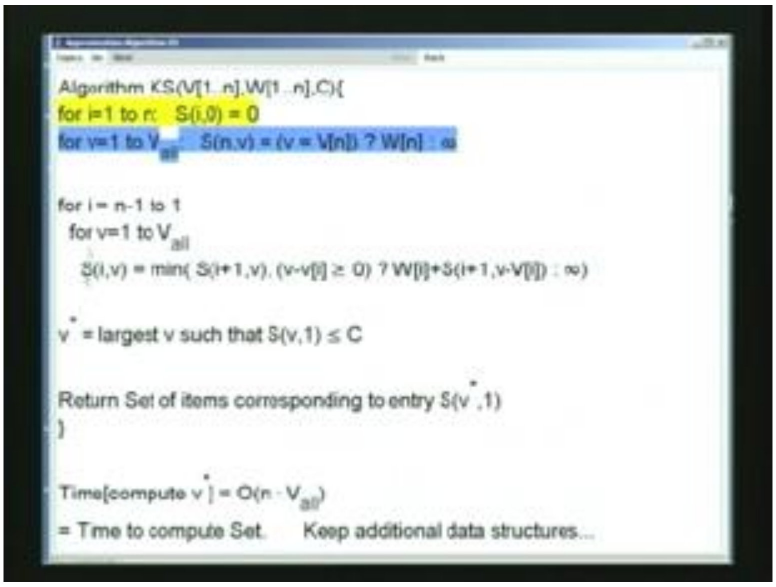
So, this row in general is going to be S n v, which is denoted, which is denoting the capacity needed to get value v, the least capacity needed to get value v using item n alone. So, you are just allowed to pick item n nothing else. So, what kind of capacities can you get? What kind of values can you get? Well clearly, if this v happens to be V of n, then you can do that using just a single item. 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 vale of v. So, let us 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 infinity, 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 infinites and if you take min of those infinities, then an infinity will crop up over here.

But, says that even this value is impossible to accomplish. So, that is sort of a nice thing. That is a kind of a nice coding that, infinity allows us to accomplish. So, basically now we have express the algorithm entirely. So, we have entries to fill. And there 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.

(Refer Slide Time: 34:46)



```
Algorithm KS(V[1..n],W[1..n],C){
for i=1 to n:   S(i,0) = 0
for v=1 to $V_{all}$:   S(n,v) = (v = V[n]) ? W[n] : ∞

for i = n-1 to 1
  for v=1 to $V_{all}$
    S(i,v) = min( S(i+1,v), (v-v[i] ≥ 0) ? W[i]+S(i+1,v-V[i]) : ∞)

$v^*$ = largest v such that S(v,1) ≤ C

Return Set of items corresponding to entry S($v^*$,1)
}

Time[compute $v^*$] = O(n · $V_{all}$)
= Time to compute Set.      Keep additional data structures...
```

So, we can write done our algorithm. So, I am going to call my algorithm KS as knapsack. It is going to take as argument. 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. So, there were those yellow entries to be filled. They were all going to be filled with 0. So, we will do that. So, these entries using this equation. Then, there were the blue entries to be filled. 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. Finally, the rest of the entries were filled using the recurrence which we derived. So, this is that recurrence. At the end of it, to get 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 column 1. And we know that, as we go as we go down this is going to be non decreasing.

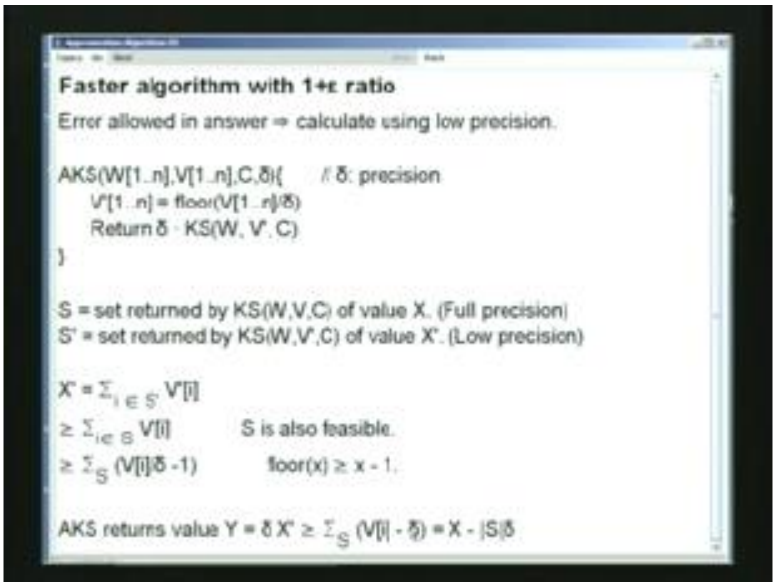
So, we can easily find the largest v. Such that, S v 1 is less than C. And that is the value, which we will call v star. So, that is the value that we are going to get. If v in fact, had a capacity of C. So, if you are just interested in the value, then we would be done at this point. However, if we 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 V all. And here, we have nested loops.

And therefore, it will take time O of n times V all. In fact, if you compute the set itself, we will have to keep some additional data structures. 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 V all 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 V all. So, that finishes the first task that we undertook. So, we now have a dynamic programming algorithm, which finishes in time n times V all. So, now we come to the approximate algorithm. Here is the important point.

(Refer Slide Time: 38:54)



Faster algorithm with 1+ε ratio

Error allowed in answer ⇒ calculate using low precision.

```
AKS(W[1..n],V[1..n],C,δ){      // δ: precision
    V'[1..n] = floor(V[1..n]/δ)
    Return δ · KS(W, V', C)
}
```

S = set returned by KS(W,V,C) of value X. (Full precision)
S' = set returned by KS(W,V',C) of value X'. (Low precision)

$X' = \sum_{i \in S'} V'[i]$

$\geq \sum_{i \in S} V'[i]$ S is also feasible.

$\geq \sum_{S} (V[i]/\delta - 1)$ floor(x) ≥ x - 1.

AKS returns value $Y = \delta X' \geq \sum_{S} (V[i] - \delta) = X - |S|\delta$

So, here we are going to be allowed an error in the answer. We are only required to get within 1 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, us to reduce the time. So, we are going to write a new procedure, a new algorithm which we are going to call an approximate case approximate knapsack or AKS.

It is going to take the same arguments as before. But, it is also going to 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 prime. So, V prime is simply going to be V divided by delta corresponding elements.

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 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. So, this is expected to do roughly the same job. 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. At the 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 is 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. Let us say, that S denotes the set returned by our original KS call. So, where we were using the actual values of V as given? 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 this by, this case without multiplication by delta for the minute. Let us say its value is X prime and this is a low precision answer.

So, what we now need to do is to relate this X prime, that we so X prime and X and so on. So, let us 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.

So, it is 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, now here is the important point. This X prime, this value is bigger than this expression as well. 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, it is some difference set.

So, what happens if it is 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, 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.

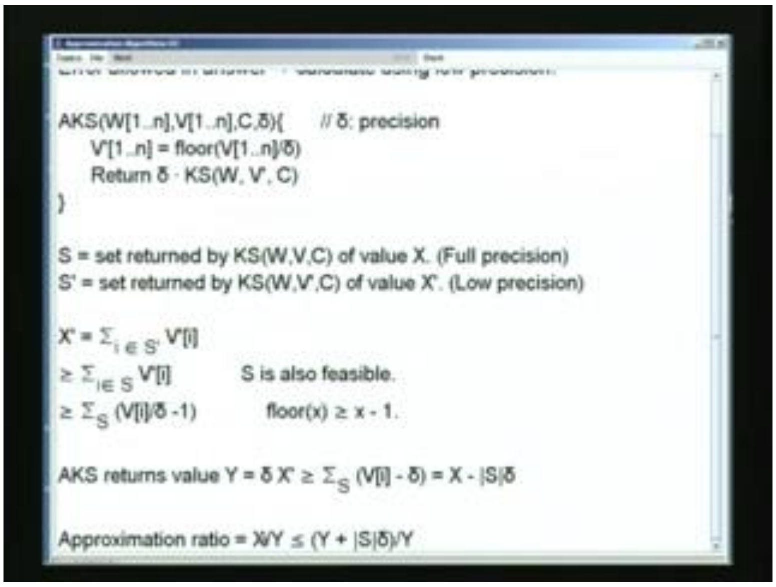
So, S is also a feasible solution. And therefore, we get this. Now finally, we observe that if I take the floor of any number, it is bigger than that number minus 1. And therefore, this is the floor of this. And therefore, this is bigger than this minus 1. So, V prime of i is the floor of this. And therefore, V prime of i is bigger than V of i upon delta minus 1 because of this. So, we have been able to relate X prime to this quantity over here.

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, let us 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. 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 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 V i. That simply X. 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 element of the set I am considering.

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.

(Refer Slide Time: 45:55)



```
AKS(W[1..n],V[1..n],C,δ){      // δ: precision
    V'[1..n] = floor(V[1..n]/δ)
    Return δ · KS(W, V', C)
}
```

S = set returned by KS(W,V,C) of value X. (Full precision)
S' = set returned by KS(W,V',C) of value X'. (Low precision)

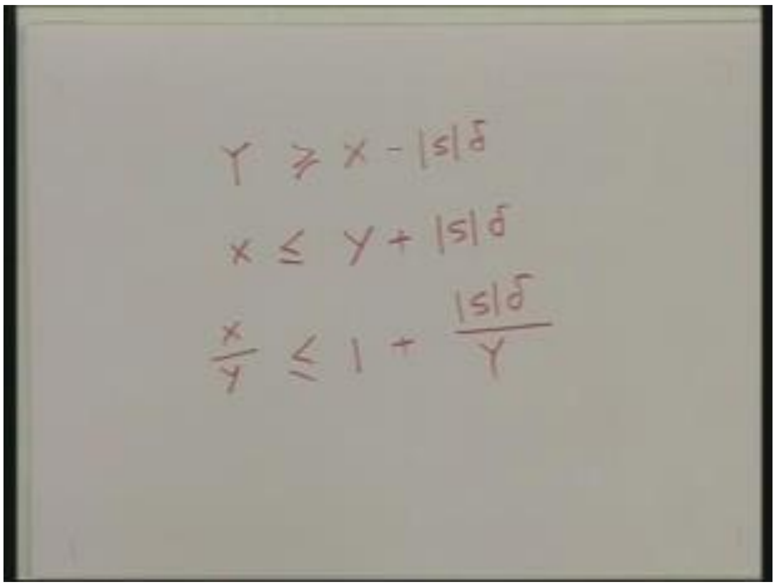
$X' = \Sigma_{i \in S'} V'[i]$
$\geq \Sigma_{i \in S} V'[i]$ S is also feasible.
$\geq \Sigma_S (V[i]/\delta - 1)$ $\text{floor}(x) \geq x - 1$.

AKS returns value $Y = \delta X' \geq \Sigma_S (V[i] - \delta) = X - |S|\delta$

Approximation ratio = $X/Y \leq (Y + |S|\delta)/Y$

So, now we are ready to evaluate the approximation ratio. What is the approximation ratio? So, it is on 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 is exactly what we are doing over here. So, we want the approximation ratio of X upon Y.

(Refer Slide Time: 46:46)



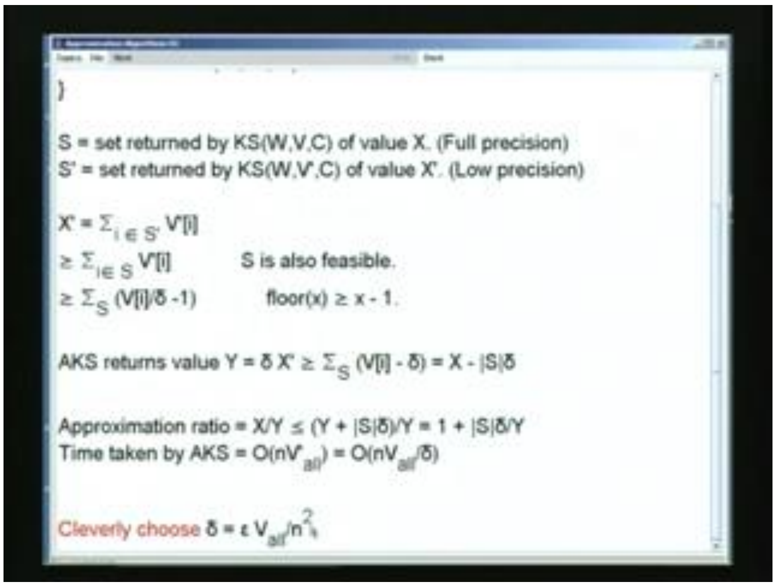
$$Y \geq X - |S|\delta$$

$$X \leq Y + |S|\delta$$

$$\frac{X}{Y} \leq 1 + \frac{|S|\delta}{Y}$$

So, let us collect 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. So, this is one inequality. 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 1 plus cardinality of S times delta upon Y. This is what we are going to get. So, this is what we finally have over here.

(Refer Slide Time: 47:34)



}

S = set returned by KS(W,V,C) of value X. (Full precision)
S' = set returned by KS(W,V',C) of value X'. (Low precision)

$X' = \Sigma_{i \in S'} V'[i]$
$\geq \Sigma_{i \in S} V'[i]$ S is also feasible.
$\geq \Sigma_S (V[i]/\delta - 1)$ floor(x) $\geq$ x - 1.

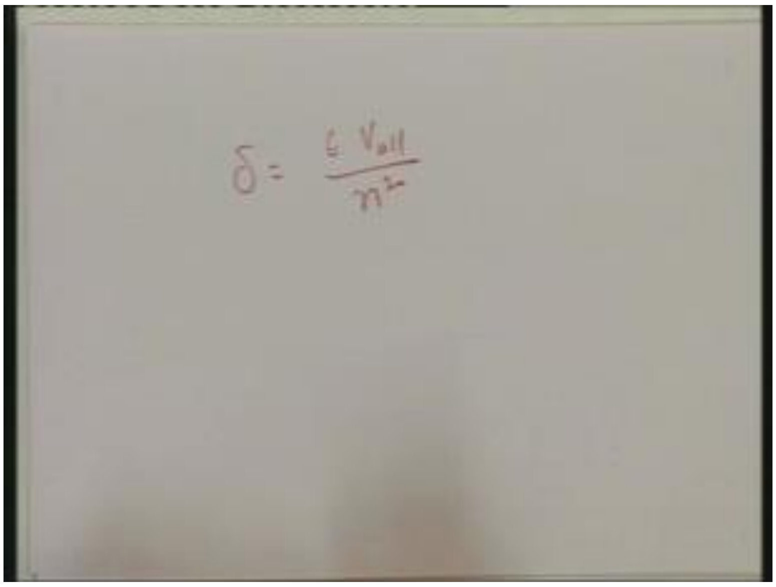
AKS returns value $Y = \delta X' \geq \Sigma_S (V[i] - \delta) = X - |S|\delta$

Approximation ratio = $X/Y \leq (Y + |S|\delta)/Y = 1 + |S|\delta/Y$
Time taken by AKS = $O(nV'_{all}) = O(nV_{all}/\delta)$

Cleverly choose $\delta = \varepsilon V_{all}/n^2$.

What is the time taken by AKS? So, it is n times V all V prime all or V prime all is essentially V all by delta. So, it is this time. 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 V all upon n square. So, let us just do that.

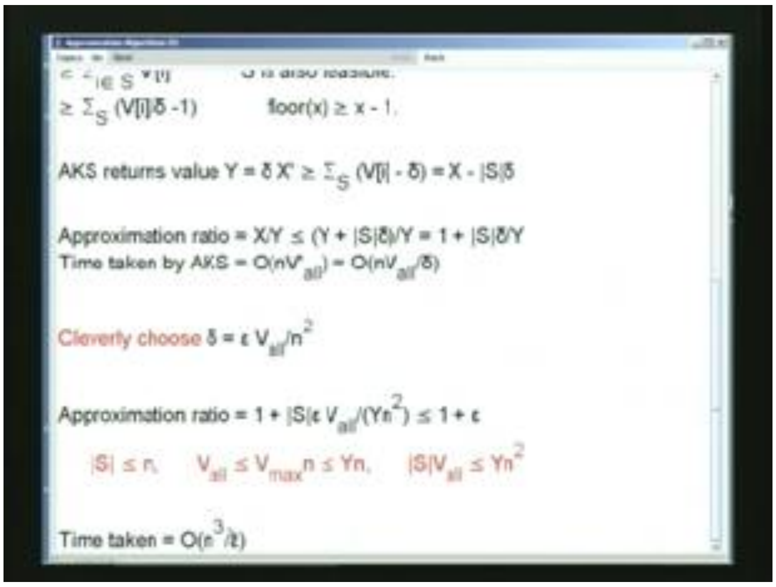
(Refer Slide Time: 48:15)



$$\delta = \frac{\epsilon V_{all}}{n^2}$$

So, we choose delta equal to epsilon times V all upon n squared. So, what does that do for us? So, first of all this was our approximation ratio. So, if we substitute into that, what do we get?

(Refer Slide Time: 48:29)



$\geq \Sigma_S (V[i]/\delta - 1)$ floor(x) $\geq$ x - 1.

AKS returns value $Y = \delta X' \geq \Sigma_S (V[i] - \delta) = X - |S|\delta$

Approximation ratio = $X/Y \leq (Y + |S|\delta)/Y = 1 + |S|\delta/Y$

Time taken by AKS = $O(nV'_{all}) = O(nV_{all}/\delta)$

Cleverly choose $\delta = \varepsilon V_{all}/n^2$

Approximation ratio = $1 + |S|\varepsilon V_{all}/(Yn^2) \leq 1 + \varepsilon$

$|S| \leq n$, $V_{all} \leq V_{max} n \leq Yn$, $|S|V_{all} \leq Yn^2$

Time taken = $O(n^3/\varepsilon)$

So, we substitute delta upon Y over here. So, we get epsilon V all. 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 1 plus epsilon. Why? So, let us see that. The first observation is that, the cardinality of S which appears over here has to be less than n.

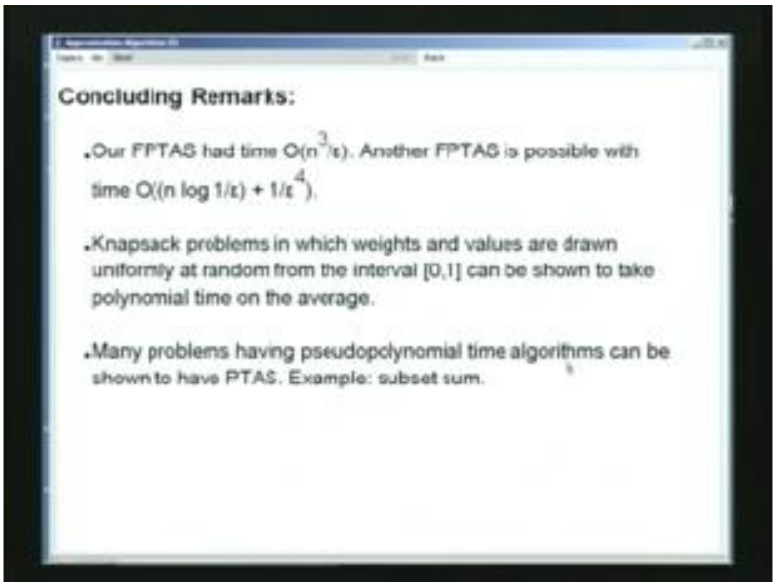
After all, what is S? It is a subset of n element. So, its cardinality has of course, to be less than n. So, that is one important observation. Then, second V all is the sum of all the elements. So, clearly it has to be less than V max times n. So, if V max denotes the maximum value if I multiplied by 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, V max cannot be bigger than Y. 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, V max times n has to be at most Y times n. So, it follows from this, that S times V all is less than Y times n squared. So, this S times this V all is less than Y times n squared.

And therefore, our approximation ratio is 1 plus epsilon. I also claim that the time taken is n cube 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 cube upon epsilon. So, what has happened? We have shown that, our approximation ratio is 1 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 cube upon epsilon. So, which is exactly what we have promised? We had promised to get an approximation scheme, in which the approximation ratio is 1 plus epsilon for every epsilon. And that we would prove that, the time taken is polynomial in n the input instance length and 1 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.

(Refer Slide Time: 51:18)



Concluding Remarks:

- Our FPTAS had time $O(n^3/\varepsilon)$. Another FPTAS is possible with time $O((n \log 1/\varepsilon) + 1/\varepsilon^4)$.
- Knapsack problems in which weights and values are drawn uniformly at random from the interval [0,1] can be shown to take polynomial time on the average.
- Many problems having pseudopolynomial time algorithms can be shown to have PTAS. Example: subset sum.

Our FPTAS had time of n cube upon epsilon. It is possible to device another FPTAS, with a different expression n log 1 over epsilon plus 1 over epsilon to the 4. So, it is very likely that in practice, this approximation. In many practical situations, this might be a better algorithm than this. But, 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 0 through 1, can be shown to take polynomial time on the average. So, in some sense the FPTAS result says that, it is easier to approximate.

And this says that, it is 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. One example is the sub another example is the subset sub problem.

Thank you.