

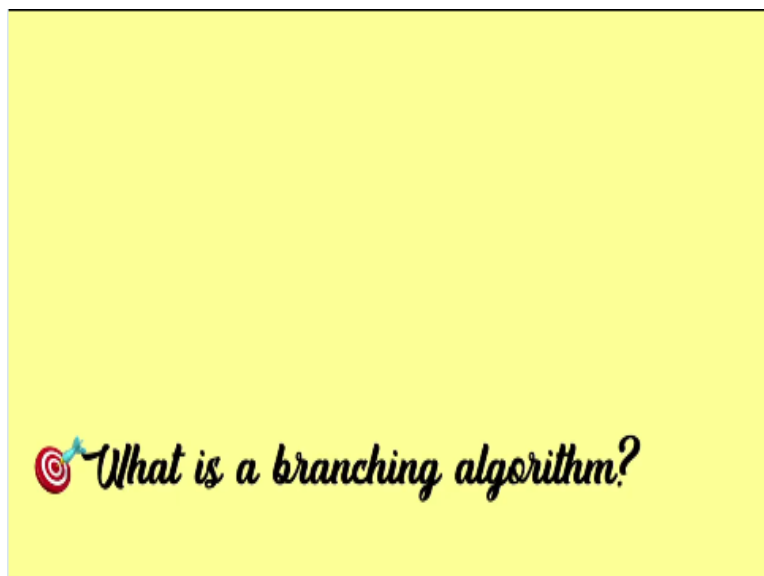
Parameterized Algorithms
Neeldhara Misra and Saket Saurabh
The Institute of Mathematical Sciences
Indian Institute of Technology – Gandhinagar

Lecture - 07
Branching Algorithms and Depth-Bounded Search Trees

Welcome to the very first lecture of the second week in this parameterized algorithm scores. I hope, you have been having a great time so far. And I also hope that you are excited to start off a brand new topic. This week, we will be exploring the theme of branching algorithms, which are also known as depth bounded search trees. And everything that we discussed in this week will be based on the third chapter of the parameterized algorithms text.

And in fact, the title of that chapter is bounded search trees. So, these terms will end up being used interchangeably as we go along. Now, in this lecture, specifically, the plan is to introduce branching algorithms using vertex cover as a running example. So, we will start off with a fairly intuitive and simple algorithm, which is also easy to analyse. And then we will try to build on it and improve the running times in a couple of different ways. So, that is the basic agenda.

(Refer Slide Time: 01:14)



But before we get started, let me talk a little bit about branching algorithms in general. Of course, by and large, we will discover branching algorithms by example, as I was just saying, but let me make some high level remarks anyway.

(Refer Slide Time: 01:19)

Let's find out!

For now, just think of these as algorithms that:

- start with an instance of the (decision) problem you want to solve,
- spawn a "small" number of "simpler" instances that collectively provide enough information to accurately determine the status of the original instance,
- and **stop spawning instances once they have become simple enough** to solve "by hand"

So, if you are familiar with recursive algorithms or backtracking strategies, you will probably find yourself right at home with branching algorithms as well. The main thing to keep in mind is that when we talk about branching algorithms, in the context of parameterized complexity, the main additional perspective is the fact that we want to bound how deep we go into the recursion Rabbit Hole by some function of the parameter that we are working with.

So, just keep that at the back of your mind. But for now, you can just think of branching algorithms as algorithms that of course, start of with an instance that they are trying to figure out. But based on this instance, they are going to create a number of auxiliary instances that they are going to recurs into. So, you can think of these as parallel universes that the algorithm is trying to simultaneously explore.

And hopefully from these instances, it gains enough information to be able to resolve the original instance that it was working with that is anyway, the incentive for generating all of these sub-problems, the hope is that they help you solve the original problem. So, every recursive algorithm must also know where to stop. So, you need some sort of a base case, which tells you that now you can call it a day and start working your way back up the recursion tree.

And the idea here is that the base cases kick in, when the instances become so small that you can resolve them by hand, or they become structurally so trivial that you can deal with them directly with some polynomial time algorithm. You do not need to recur any further. Now

from here, you can probably tell that the running time of such an algorithm is essentially governed by 2 factors.

(Refer Slide Time: 03:19)

The complexity of such algorithms is determined by:

- how small is "small" 🙌

The smallness of the number of instances we generate determines the breadth of our search;

- how quickly we can "simplify" 🙌

while the quickness with which we are able to simplify determines the depth of our search.

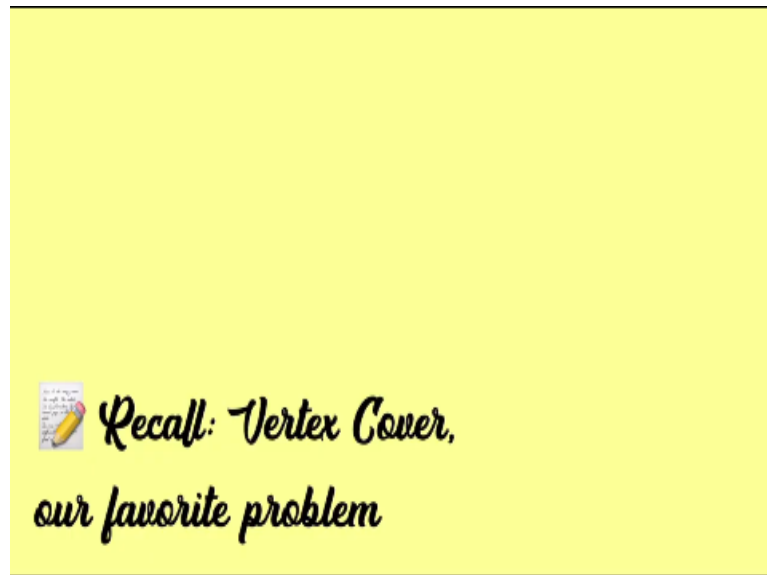
The first is how small is small. Remember, we do not want to generate too many recursive sub-instances, because that is an expensive thing to do. And the other aspect is how quickly can you hit the base case, because that is your finish line. And you want to get there quickly. So, remember, we said that you hit the base case, when your instances become really simple. And hopefully, the recursive sub-problems that you are generating at every step, they are simpler versions of the problem that you started with.

They are typically either smaller, or somehow they have been structurally, you know, simplified in a way that you can see that there is an improvement, you are getting closer and closer to the base case with every step. If this is not happening, then you probably have a dangerous algorithm on your hands. And you should revisit what your algorithm is, in fact, doing.

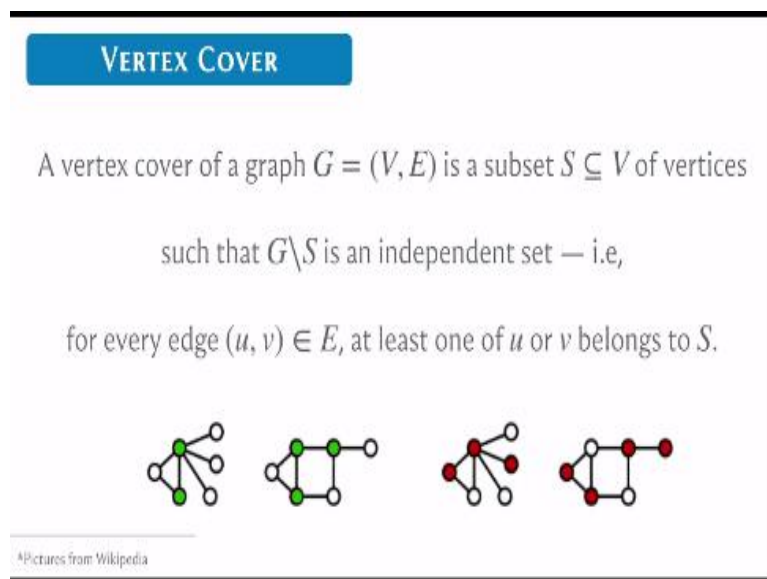
So, the smallness of the number of instances that you generate at every step, we think of that as the breadth of our search tree that is how much you expand at every step. And on the other hand, the rate at which you simplify your instances to hit the base case, you think of that as the depth of your search tree. So, again, if all of this is sounding a little bit abstract at this stage, do not worry about it, it will become more explicit as we work through a couple of examples.

And we actually see a concrete search tree corresponding to a recursive algorithm for the first time, but hopefully this still builds up some intuition for what to expect. And as I said, we will revisit this discussion after we have gone through the vertex cover algorithms at the very end of this discussion. So, now let us get started with vertex cover.

(Refer Slide Time: 05:06)



(Refer Slide Time: 05:10)



This by now should be a familiar problem. But just to be sure, let us recall the definition. For a graph G , a vertex cover S is simply a subset of vertices, which contains at least one endpoint of every edge. Equivalently, you could think of it as a subset of vertices whose removal makes the graph empty in terms of edges. So, $G - S$ is an independent set.

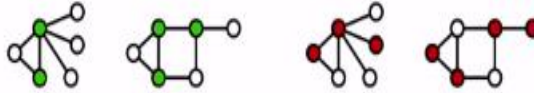
(Refer Slide Time: 05:34)

VERTEX COVER

Input: A graph $G = (V, E)$ and a positive integer k .

Output: Yes if and only if G has a vertex cover of size at most k .

Parameter: k .



*Pictures from Wikipedia

The natural computational question associated with vertex cover is a minimization question. So, you want to find a smallest vertex cover. And of course, we normally work with the decision version of the problem, where we have been given an explicit budget, which we will call k , which also doubles up as our parameter. And the question is, if G has a vertex cover of size at most k . And typically, our quest is to find FPT algorithms for vertex cover parameterised by k .

(Refer Slide Time: 06:09)



So, with all that said, let us actually try and devise a first branching algorithm for vertex cover.

(Refer Slide Time: 06:12)

"Branching" on an edge.



Suppose we are dealing with an instance (G, k) .

Let (u, v) be an edge in G .

(G, k) is a YES-instance
if and only if
at least one of $(G - u, k - 1)$ or $(G - v, k - 1)$
is a YES-instance.

So, let us assume that our graph has at least one edge, because if it does not, then we really do not have any work to do. And we can walk home happy. And that is the scenario that we want to get to that will be in some sense, the kind of base case that we are looking for a graph where there is no work left to do. But as long as this is not the case, there is at least one edge that is available in the graph.

Now, based on the definition that we just went over, can you think of an exhaustive approach to finding a vertex cover of size at most k based on just this edge? So, one thing that you do know about any vertex cover is that it must intersect this edge somewhere, you do not know where. But that is the point of an exhaustive approach. It is to say that when you do not know, you try all possibilities.

So, this is a good place to pause this video and think about how you would come up with a recursive algorithm to check if G has a vertex cover of size at most k or not that is anchored around some arbitrary edge in the graph. So, the graph is non empty, you pick your favourite edge, it could be any edge in the graph, let us call it uv . And based on this edge, how would you think of recurring? So, think about that and come back when you are ready.

Hopefully you had a chance to think about this a little bit. And let us continue and actually make our first branching algorithm explicit. So, suppose this instance that we are dealing with, let us denote it as G, k . And this edge, of course that we already have is uv . And the thing to observe is that G, k is a yes instance of vertex cover, if and only if at least one of these 2 instances that you see here is a yes instance.

And this is just based quite directly on the definition of vertex cover. So, what we want to say is that either $G - u$ has a vertex cover of size at most $k - 1$, or $G - v$ has a vertex cover of size at most $k - 1$. If both of these are no instances of vertex cover, then there is no hope for G to have a vertex cover of size at most k . Because suppose to the contrary that indeed, so if G has a vertex cover of size at most k , let us call it S .

We know that either u belongs to S or v belongs to S . This is not an exclusive or it is possible that both of these vertices belong to S . But for us, it is enough to know that at least one of these vertices belongs to S . So, without loss of generality, let us say that u belongs to S . If u belongs to S , then let us consider the subset $S - u$, $S - u$ is going to be a perfectly valid vertex cover for the graph $G - u$.

Because if it was not and there was an uncovered edge in $G - u$ that would be an uncovered edge relative to S in the graph G , contradicting our assumption that G had a vertex cover in S . So, we know that if G had a vertex cover of size at most k , then surely one of these 2 instances work out to being the yes instance. Conversely, if at least one of these 2 instances work out to being a yes instance, it does not matter which one.

So, this time, let us say for example that $G - v, k - 1$ is a yes instance. Then we can use this fact to argue that G, k is also a yes instance. So, let us say that the fact that $G - v, k - 1$ is a yes instance is witnessed by some vertex cover S prime. Now, consider the set S prime union v and let us call this S , you should be able to convince yourself quite easily that S is a valid vertex cover for G .

In particular, every edge that is incident to the vertex v is covered by v , because we explicitly included it in the vertex cover and any other edge in the graph is covered by S prime because any other edge is also present in the graph $G - v$ and S prime was a valid vertex cover that. So, using this fact, we can come up with a very natural recursive algorithm for vertex cover.

We say when we are handed over the instance G, k , what we are going to do is invoke our algorithm on the instances $G - u, k - 1$ and $G - v, k - 1$ and we wait for a response. And if both of the responses are no, then we say no. But if at least one of the responses is yes, then we say yes. This almost completes the description of our branching algorithm.

The one thing that is missing, of course, is the base case, remember, we do want our algorithm to stop that is very important. So, when is this algorithm going to stop? Remember, you do have to pick an edge to be able to meaningfully branch in this branching algorithm. So, one place where you might get stuck is if your graph becomes empty.

If your graph becomes empty, while you still have some budget left, or if your graph becomes empty, exactly the same time when the budget becomes 0, then of course, you can stop and say that you are done. And you can simply return yes, at this point. On the other hand, if you know k becomes 0 and you still have edges in the graph remaining, then at this point, you know that you are dealing with a no instance.

You do not have to investigate any further. You do not have any budget left, you still have work to do. And now, you are in an impossible situation. So, at this point, you can simply return no.

(Refer Slide Time: 12:14)

"Branching" on an edge.

u — v

Suppose we are dealing with an instance (G, k) .
Let (u, v) be an edge in G .

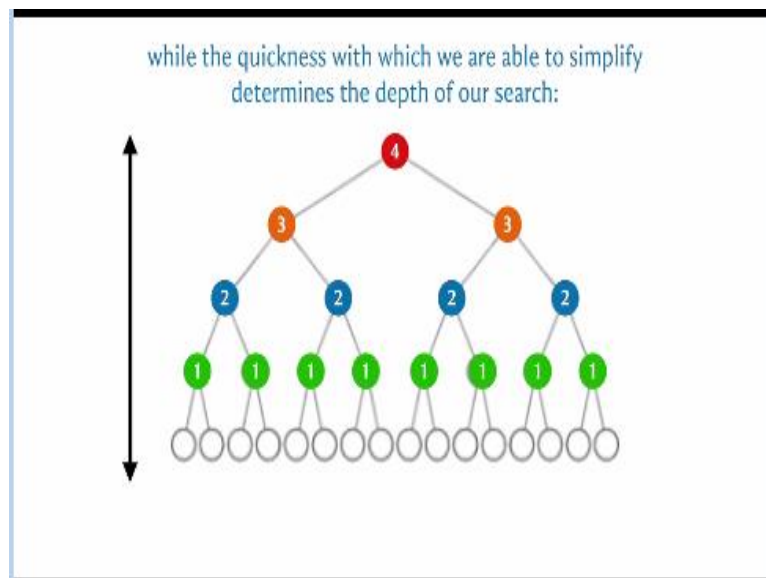
Also, if $k = 0$, (G, k)
is a YES-instance if and only if:

So, this essentially becomes your base case. So, when you hit a budget of 0, then you know that you have to say yes, if and only if the graph at that point is empty. So, this essentially completes the description of our algorithm. So, as long as k is nonzero and the graph is non empty, we have a way to make progress. But the moment, we falsify either of these criteria, so either we run out of budget or we run out of edges, then we know exactly what to do. We do not need to venture out any further.

And we can stop, we call it a day. So, now that we have completely described the algorithm and we are hopefully convinced of its correctness as well. We can now start talking about the analysis of the running time. So, remember, we said that the expense of a branching algorithm is essentially governed by the depth and the breadth of the search tree that is naturally associated with it.

And further, we said that the breadth has something to do with the number of instances that we generate at each phase. And the depth has something to do with how quickly are we able to reach a base case. So, let us explore these 2 factors in the context of this specific algorithm that we have just described by taking a look at what the search tree looks like.

(Refer Slide Time: 13:38)



So, notice that to begin with, you have the instance G, k . And let us say the parameter k is 4, just as an example. Now, what we do from here is; we generate 2 sub instances. And in both of these instances, notice that the parameter decreases by 1, so, we go into $G - u, k - 1$ and $G - v, k - 1$ where uv are the endpoints of some edge.

Notice that I am assuming that the graph was non empty to begin with, otherwise, there is really nothing left to discuss. Now from here, each of these 2 sub-instances will generate 2 sub-instances of their own, but with a further reduced parameter value. So, notice that k will go down from 3 to 2. And now each of those 4 sub-instances will generate to serve instances of their own.

And this leads to a total of 8 instances in the 4th layer. But notice that by now the parameter value has reached 1 and you are now at the brink of extinction. Or I should probably say termination, because in the very next step, the parameter values are going to drop to 0. And you can stop with 1 vertex or another immediately right there. Now, also notice that this may not be reflective of an actual run on an actual instance, because it is possible for instance that you run out of edges at some places.

And this search tree in many parts may end up truncating early. But the point here is to say that this is the worst case scenario, at the most, the number of nodes that the search tree could have, is going to be only so much. It will never go deeper than this. And it certainly has no reason to go wider than this, just by the structure of how the algorithm works. So, notice that we are generating 2 recursive calls at every stage. So, that is governing the breadth.

And the fact that the parameter is dropping by 1 at every stage is governing the depth. So, that combined with the fact that we stopped when the parameter hits 0 if we did not have that base case. And let us say, as a bizarre hypothetical, let us say that we only knew how to deal with instances where the value of k is -25 , then your branching algorithm would have to be analysed differently.

And your surgery could have potentially gone deeper. But it is usually a convention that we stop when some measure that we can associate with the instance, reaches a stage where its value is 0. So, the measure being 0 is a way of saying that your instance has been simplified to the point where you do not need to do anything more. And for this example, our measure of simplicity is simply the parameter itself that already tells us.

It is an indicator for how much progress we are making. So, with all that said, finally, what is the overall running time? You can probably see that the work done by the branching algorithm is proportional to the size of the search tree or in particular, the number of nodes in the search tree. In turn, the number of nodes in the search tree is essentially proportional to the number of leaves in the search tree, because the number of internal nodes is bounded by the number of leaves.

So, what we will typically do is basically focus on bounding the number of leaves in the search tree. If we do that, then the overall running time is going to be this estimate that we

have for the number of leaves with some polynomial overhead, because we typically assume and this is usually the case that it takes a polynomial amount of time to generate the sub instances. And it is a polynomial amount of work at the base case.

(Refer Slide Time: 17:49)

"Branching" on an edge.

u — v

Suppose we are dealing with an instance (G, k) .
Let (u, v) be an edge in G .

RUNNING TIME

$$T(k) \leq T(k-1) + T(k-1) = 2T(k-1) \leq 2^k$$

So, with those functions in place, the running time is essentially governed very naturally by this recurrence, where we use T of k to denote the maximum number of leaves that we will have in a search tree for this algorithm, which starts off with an instance whose parameter is k . So, you can hopefully see that this recurrence holds. This is essentially coming from the form of the recursion calls, we generate 2 instances where the parameters are both $k - 1$.

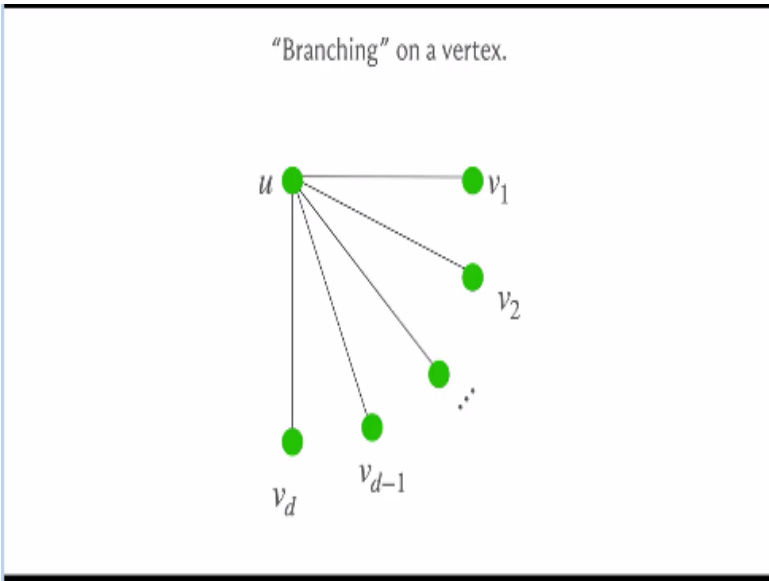
And so, that is why we have $T(k-1)$ plus $T(k-1)$. And the form of this recurrence is simple enough that you can simply solve it by expansion, to see that it works out to 2^k . This was probably quite predictable already, visually from the search tree that we saw just a moment ago. But it is also useful to work it out.

(Refer Slide Time: 18:43)

🏆 Improved branching:
focus on vertices instead of edges

But now, let us think about coming up with a better branching algorithm. So, instead of branching on an edge, you are going to be coming up with a strategy that is centred around branching on a vertex instead.

(Refer Slide Time: 18:57)



So, let us consider what this might look like. So, suppose you have a vertex u , whose neighbours are v_1, v_2 and so on, up to v_d . And again, let us assume that d is at least 1. In fact, we would perhaps like to assume even that d is at least 2, for reasons that will become clear in a moment. But d is at least one is the same as saying essentially that there is at least 1 edge in the graph.

I am trying to push that a little more to say that let us suppose that there is at least a vertex of degree at least 2. And let us see if you know there is something useful that we can do with

such a vertex. So, what does it mean to branch on a vertex? It means that we want to somehow come up with an array of exhaustive possibilities anchored around this vertex in some sense, perhaps we want to explore what happens when a solution includes this vertex, as opposed to when it does not include this vertex.

So, remember that, if you were working with a yes instance, then there is at least one solution. And the exhaustive possibilities that might be vertex exploring are the following. The first possibility is that this solution does include the vertex u . The second possibility is that the solution does not include the vertex u , notice that these are 2 possibilities that are completely exhaustive.

And in one of them that is a very natural recursive instance to consider, which is very similar to what we have already discussed, when we were branching on an edge. But in the other situation, it is not nearly as obvious as the first one. So, again, I would encourage you to take a moment here to pause and think about what is going on, before you come back and continue this discussion with me.

So hopefully, you realised that the easy case is when the solution that you have in mind chooses to include the vertex u because this is just like, including one of the endpoints of the edge that we were branching on just now. So, if the universe that you want to go into is exploring the possibility that there is a solution that picks the vertex u , then in this universe, we know that $G - u_{k-1}$ must be a yes instance. So, we explore that.

And again, here, you can see that your parameters dropping by 1, so hopefully, there is some sense in which you are making progress. But what about the situation when you are exploring the possibility that the solution does not contain the vertex u ? So, here, it is not completely clear, what graph we should generate for the recursion to consider. We could say that, we are going to label this vertex in some special way.

And say, you know, let us just avoid this vertex, we have pre-determined that we are not going to include it in the solution. But then it is not clear, what is the sign of progress? You could say that the number of unmarked vertices has reduced. But if you pursue this thought a little bit further, you will see that it is not a very useful measure, because the number of unmarked vertices to begin with could be as large as n .

So, the depth of your branching does not get bounded meaningfully as a function of the parameter. So, if this sort of a data did not make a lot of sense, do not worry about it. Let us just think about what is a useful thing to do here. So, let us go back to the drawing board. If the solution does not pick the vertex u , is there something that it is forced to do?

You might remember from the kernelisation lectures that when you were exploring the higher degree reduction rule, you saw that if you do not pick a high degree vertex, then essentially, you cannot even be a solution. And the reason for that was that if you do not pick this vertex, then who covers the edges that are incident on it? That is a great question to ask right here as well. So, if the solution does not pick the vertex u , which vertex is responsible for covering the edge uv .

Well, there is not much of a choice left it has to be v . For the same reason, you must also take v_2, v_3 and so on, all the way up to v_d . So, in case you decide not to pick the vertex u , you must in fact, pick all of its neighbours. So, that is going to be our second universe, a second world of possibility, it is the graph $G - u$ – the closed neighbourhood of u with the parameter $k - d$ – the degree of u , because sorry, $k - d$ – the degree of u , because we know that degree of u many vertices have been forced into a solution.

Any solution that does not pick u is compelled to pick all of its neighbours. And we want to leverage this fact, in our branching algorithm.

(Refer Slide Time: 24:21)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

Further, suppose $d \geq 2$.

(G, k) is a YES-instance
if and only if
at least one of $(G - u, k - 1)$ or $(G - N[u], k - d)$
is a YES-instance.

So, the formal statement that you want to prove is that (G, k) is a yes instance if and only if at least one of $(G - u, k - 1)$ or $(G - N[u], k - d(u))$ is a yes instance. One of these should work out to being yes instances. Now, the explanation is what we have, you know, just discussed a moment ago.

(Refer Slide Time: 24:50)

- ▼ The forward direction.
 - Suppose (G, k) is a YES-instance.
 - Then G has a vertex cover, say S , of size at most k .
 - If $v \in S$, then $(G - v, k - 1)$ is a YES-instance.
 - If $v \notin S$, then $N(v) \subseteq S$.
 - Therefore, $(G - N[v], k - d(v))$ must be a YES-instance.

But if you want a slightly more notational way of saying it or just a more formal sequence of arguments, I am going to share with you a couple of slides where we make this explicit.

(Refer Slide Time: 25:02)

- ▼ The reverse direction.
 - ▼ Suppose $(G - v, k - 1)$ is a YES-instance.
 - Then $G - v$ has a vertex cover, say S , of size at most $k - 1$.
 - Notice that $S \cup \{v\}$ is a vertex cover of G of size at most k .
 - ▼ Suppose $(G - N[v], k - d(v))$ is a YES-instance.
 - Then $G - N[v]$ has a vertex cover, say S , of size at most $k - d(v)$.
 - Notice that $S \cup N(v)$ is a vertex cover of G of size at most k .
 - Therefore, in both cases, (G, k) must be a YES-instance.

But I will not go over this verbatim because I think we have discussed the spirit of these arguments. But if you want to take a look, then this would be a good place to pause. Now, the next thing that I want to consider is the running time. So, when we proposed branching on

vertices instead of edges, the advertisement was that we are going to get a superior running time. So, let us start thinking about this.

(Refer Slide Time: 25:27)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

If $d = 1$...?

RUNNING TIME

$$T(k) \leq T(k - 1) + T(k - d) \leq ?$$

So, notice that the algorithm itself branches into 2 possibilities. In the first, it picks the vertex; in the others, it picks the neighbourhood of the vertex. So, the value of the parameter k drops by 1 in the first branch and that is always fixed. And in the other branch, the drop in the parameter depends on the degree of the vertex that you are branching on. So, if the vertex had 150 neighbours, then your measure will suddenly dropped by 150.

But if the vertex has only one neighbour, then your measure is going to drop by 1. And now that does not look very good. Because if you happen to be branching on a vertex of degree 1, then this recurrence is going to look like exactly the same recurrence that we had when we were branching on edges. So, we want to avoid the situation of branching on vertices of degree 1.

Because for all, you know, if you if it is not your day and if you are unlucky, you might keep choosing vertices of degree 1 to branch on and your search tree is going to look very much like the one that we had before, which is, of course not ideal. So, for as long as possible, you know, we want to branch on vertices that are not degree 1 vertices, they have hopefully more substantial degree.

(Refer Slide Time: 26:55)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

Further, suppose $d \geq 2$.

What if G has no vertices of degree at least two?

How will we proceed?



And the question is, can we always find vertices whose degree is at least 2, so I am, you know, I am just going to assume that the graph always has a supply of vertices whose degree is at least 2. And that naturally prompts the question of what happens when we run out of search vertices. So, what happens when the degree of every vertex is at most 1? There are no degree 2 or more, no vertices whose degree is at least 2 to branch on, it might seem like we are stuck at this point.

But it is actually a blessing in disguise, because if you think about it, what do graphs look like when they do not have vertices whose degree is at least 2. This means that every vertex has degree 0 or 1, which means that every connected graph, sorry, every connected component in this graph, must look like either an isolated vertex or an edge, there really is not any scope for anything else to be present in this graph.

But notice that these graphs are therefore very simple. And remember, we said that we hit a base case, whenever we hit a situation that we can resolve directly. So, we do not have to necessarily always wait for k to become 0 to say that we have arrived at the base case. So, in this case, we have a situation where the graph is so simple that we can just resolve it directly.

So, we just count the number of edges in the graph, these edges will necessarily constitute a matching. So, we just check if k is at least M or not. So, if k is less than m , then we say no. And if k is at least 10, then we say yes. And that is pretty much the end of the story. So, as long as, we have a degree to vertex to branch on, we branch on it. In fact, we can always

choose as a heuristic to branch on the vertex that has the largest degree because notice that that will give us the fastest drops in at least the second branch.

So, even though the left branch is going to progress slowly with k reducing by only 1 at each step. Now, your search tree is kind of skewed. It is not this nice symmetric, complete binary tree kind of picture that we had earlier. So, you have the left branch progressing slowly and it will go k levels deep and the worst case, but the right branch is dropping off much faster and you will get some weird shape which I will not even attempt to draw.

(Refer Slide Time: 29:32)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

Further, suppose $d \geq 2$.

RUNNING TIME

$$T(k) \leq T(k-1) + T(k-2) \leq 1.618^k$$

But you could write out the recurrence and the recurrence is going to be again in the worst case, T of k is certainly going to be no more than T of $k-1$ plus T of $k-2$. Because with this stronger base case, you have ensured that the drop of k in the other branch is guaranteed to be at least 2 because you stop when you run out of degree at least 2 vertices to branch on.

Of course, it is possible that your search tree is a whole lot better than this. You might have good supply of vertices that have higher degree. So, all of that is even better in the actual run of the algorithm. Once again, this is just a worst case upper bound, because for all, you know, maybe you only have vertices of degree 2 to branch on, in which case, this bound might even end up being tight.

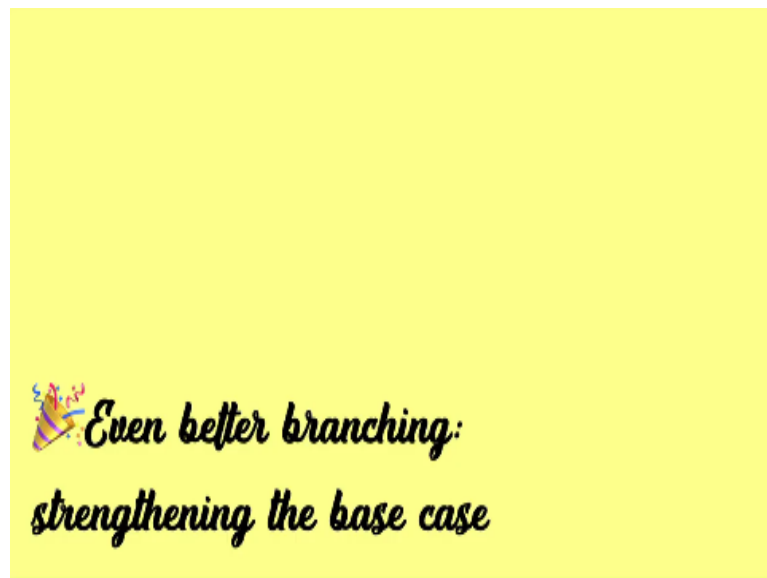
Now, this recurrence solves to 1.61 to the k , which is better than 2 to the k clearly. But maybe you are already thinking about how did we get this number. Some of you may be familiar with this recurrence from, you know, before it looks familiar, it probably reminds you have

the Fibonacci recurrence. And if you have seen a closed form for the Fibonacci recurrence, this constant might also be familiar.

So, you might be able to solve this specific recurrence from first principles once again. But this is not something that I want to worry about right now. So, in the next lecture, we will talk about a general method for how we solve such simple linear recurrences. So, this is an issue that I am going to postpone for now. And basically, I want to move on to addressing the perpetual question of, can we do better.

So, we have progressed from 2 to the k to 1.618 to the k . And now, we want to be a bit greedy and ask ourselves if we can improve this even further.

(Refer Slide Time: 31:44)



So, let us think about what is a natural place for extra leverage. So, if we go back to the algorithm that we had earlier, notice that it is the second branch that is looking really tempting, because this is where the measure drops by the degree of the vertex that we are branching on. So, just by ensuring that we always are able to branch in a vertex of degree, at least 2, we managed to ensure that the parameter and this branch are dropped by at least 2 at every step.

Now, what if we are able to ensure that instead of always being able to find a vertex of degree at least 2, we are able to find a vertex with degrees, at least 3 that would mean that you can actually guarantee a drop of 3 every time you get into the right branch. And I do not mean the

right branch in the sense of the correct branch, of course, both branches are equally correct. Or it does not even make sense to talk about the correctness of branches.

What I mean is really the second branch, so let us make sure that we clarify that. So, in that branch, the measure will drop by at least 3, if we can always be assured that we will find a vertex of degree at least 3 to branch on. So, this seems like a tempting consideration.

(Refer Slide Time: 33:09)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

Further, suppose $d \geq 2$.

What if G has no vertices of degree at least three?

How will we proceed?



So, let us again ask ourselves, you know, what would happen if we do not have any search vertices to branch on. So, if G has no vertices that have degree at least 3, it means that every vertex has degree at most 2. So, I am going to leave this as a little bit of food for thought kind of an exercise. It is something that requires an approach that is very similar to what we did before.

We said earlier that if you know the maximum degree of a graph is at most 1, then it is essentially a bunch of isolated vertices and isolated edges. And it is something similar that is happening here. If the maximum degree of the graph G is at most 2, what can you say about every connected component of G ? Does it have some very simple structure? And if yes, how can you use your understanding of this structure to resolve the problem of vertex cover and polynomial time?

I mean that is the main idea. The idea is to say that you know, once you get to this stage, your graph is simple enough that you can think of this as a base case, you can just resolve it directly. And if this is not the situation you are in, then it is also a win because you will be

able to find a vertex whose degree is at least 3 to branch on and then you get the kind of running time that you were hoping for.

(Refer Slide Time: 34:35)

“Branching” on a vertex.

Suppose we are dealing with an instance (G, k) .

Let u be a vertex in G whose degree is d .

Further, suppose $d \geq 2$.

RUNNING TIME

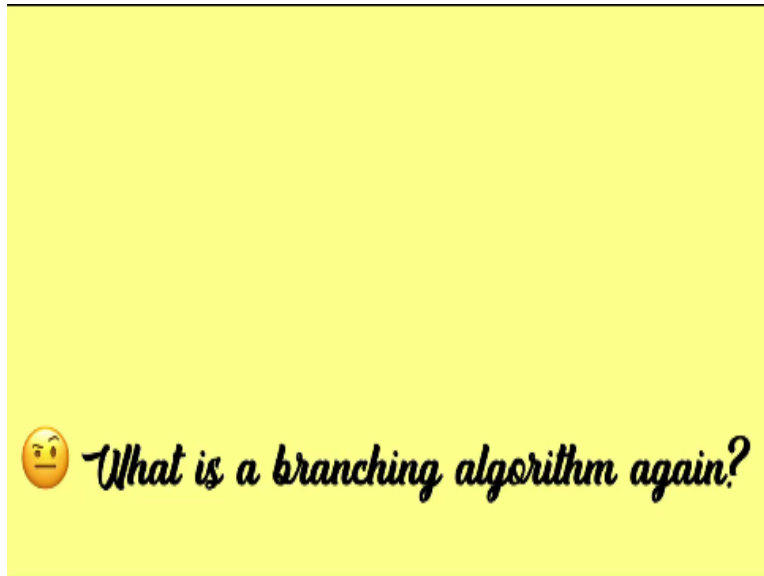
$$T(k) \leq T(k-1) + T(k-3) \leq 1.4656^k$$

So, again, I have thrown in a number here as the claimed solution for this recurrence. And once again, I will remind you that we are postponing the issue of how do we get to this number but the one thing that is worth noting is that this is a number which is smaller than the number that we had before. So, previously, we had 1.6 something to the k . This is a 1.4 something to the k .

And of course, in some sense, you should expect this because these were 2 very similar recurrences with the only difference being that k is dropping faster than it was dropping before in one of the branches. So, in that sense, this is not a surprise. But where do you get the precise number from that is something that we will, again, come back to later. So, hopefully, you will be able to work through the details of the base case yourself.

And so with that, I would say that we have concluded our discussion of branching algorithms for vertex cover.

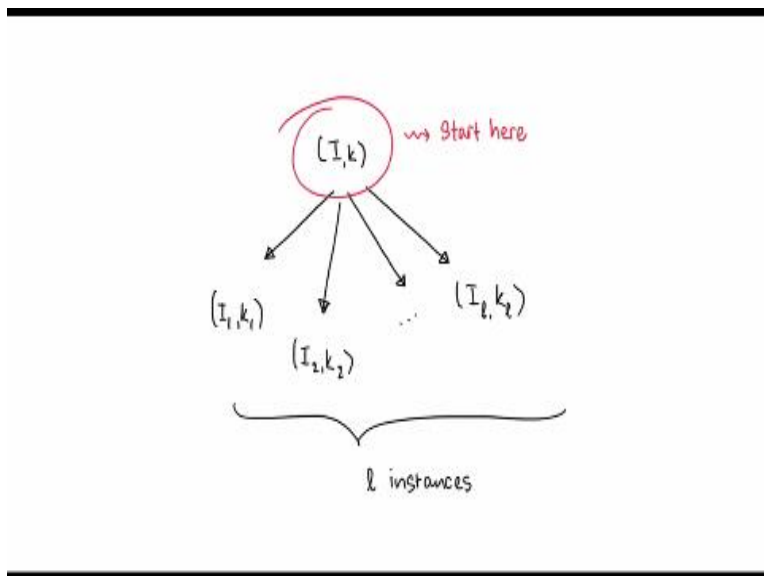
(Refer Slide Time: 35:41)



Before we wrap up to let us just go back to the question that we posed at the start of this discussion, which was what is a branching algorithm. Of course, there is not a hard and fast definitions. So, I made some vague noises back then in the beginning. And I am going to basically make those noises again, but this time with some notation and the hope that it will be easier to appreciate.

Now, that we have actually seen a couple of concrete examples of branching algorithms. The idea of this introductory discussion is to just convey some sense of the broad defining characteristics of this class of algorithms. So, that when we look at them more and more in the coming lectures, you roughly know what to expect in terms of the overall template. So, let us dig into this a little bit.

(Refer Slide Time: 36:27)



So, we begin with, of course, an instance of some parameterised problem, let us call it I, k . And bronchial algorithm will typically generate a number of sub-instances. Let us denote them as I_1, k_1, I_2, k_2 up to I_ℓ, k_ℓ . So, let us say we generate these ℓ instances. And there are 2 aspects, if you remember that we want to keep in mind. The first is that these sub-problems should be useful in terms of helping you solve the original problem.

And the other aspect is that the running time of this kind of a recursive algorithm is governed by both ℓ which is the number of instances you generate and some concept of the speed with which you are able to reach the base cases, which are your sciences for termination. So, let us try to formalise both of these ideas just a little bit.

(Refer Slide Time: 37:21)

1. Every feasible solution S of I_j has a corresponding feasible solution in I .
Let's identify the latter by $h_j(S)$.
2. The set: $\{h_j(S) \mid 1 \leq j \leq \ell \text{ and } S \text{ is a feasible solution for } I_j\}$ contains at least one optimal solution for I .
3. The number ℓ is small (bounded by a constant, for instance).
4. We can associate a "measure" $\mu(\cdot)$ with every instance.
5. For all $1 \leq j \leq \ell$, we have $\mu(I_j) \leq \mu(I) - c$ for some constant $c > 0$.
6. Once the μ -value drops below a certain threshold (typically a small positive constant), we can stop branching further 😊 — the instance is now "easy to solve".

So, the first couple of things we want to say is that if you have solutions in the sub-instances, then you have some natural way of associating these solutions with solutions in the original instance that you started with. So, this is like saying, for example that if S prime is vertex cover in $G - Vv$, then S prime union v is a vertex cover and the instance G . So, that is the kind of association we are talking about.

The other thing is that if you look at all your sub-problems and the solutions that they have and you see what they map back to in the original instance, so that is the space of all solutions that you get by association from sub-problems, then the thing that you are looking for should be here somewhere, okay? So, every instance, of course, has an optimal solution. And if that optimal solution cannot be recovered by association from one of the sub-problems, then these sub-problems are not very useful, right?

So, the thing that you are looking for should be sitting in the space of solutions that you get from all of your sub-problems. So, this is, in a sense, just trying to tell you that your sub-problems are sufficient for you to figure out, you know, whether your instance has the kind of solution you are looking for or even if you are solving the optimization version, the sub-problems are enough for finding the optimal solutions.

So, this is about the sub-problems being useful. And this kind of covers the correctness of such an approach that covers the exhaustiveness of such an approach. The next a few points are related to the running time. So, the first thing is about the number of instances that we generate at every stage. So, we denoted this by l . And of course, we want this to be bounded in some way.

In particular, if this is not bounded as a function of k , let us say you generate and by 4 recursive sub instances, then of course, it may still be a correct and valid algorithm, but you would not expect the running time to be FPT. So, you want the number of sub-instances you will generate to be hopefully bounded by a constant or at least some function of the parameter because that is going to play into your running time in a significant way.

Typically, it will be the base of the exponent that in the expression that measures the size of the search tree. The other thing that the size of the search tree is governed by, is, of course, the depth which you can think of as how much time do you need to reach the base case. And this is captured by the notion of a measure, which is a very fundamental concept when it comes to branching algorithms.

So, a measure is just a function that associates a number with every instance, which in some sense captures the complexity of the instance. And the idea is that if this measure is very small, typically, let us say that the measure is 0, then the message that it is sending you is that now this instance has become really simple and it can be solved directly. So, it is a stop signal for the algorithm. This is where you can break out and you know, you could think of this as the base case.

So, when the measure hits 0 that is when you can stop that is the event that you are looking forward to. And let us say in the beginning, the measure is bounded by some function of k ,

then our main task is to ensure that we design our sub-problems in such a way that the measure actually decreases. So, this is what we mean here, by the fifth point that the measure decreases strictly.

So, that way, what happens is that as you progress through the layers of your search tree, you know that you want keep going forever, you know, you have to stop when the measure becomes 0. And because your measure is strictly decreasing at every step and because it is bounded by some function of the parameter to begin with, you know that the depth of your recursion is also bounded by some function of your parameter.

Of course, this is a very coarse way of putting it when you are actually working with a specific branching algorithm, you do have to get into the nuances of how fast your measure is reducing. And you may have to compare various trade offs and things like that. But there is no point in talking about that too much, because that is something that becomes clear only when you are working with actual examples.

For now, it is enough to know that there needs to be some notion of a measure, which helps you track your progress. To begin with, the measure itself should be bounded; it must decrease strictly with each step of recursion. And if the measure becomes 0, then that should correspond to some sort of a stop signal. This ensures that your distance from the start point to the stop point is bounded by some function of k .

So, at a high level, just remember that the first 2 points formalise the idea of the branching being exhaustive. And the last 4 points capture the running time of the branching algorithm and ensure that it is bounded in a meaningful way for us.

(Refer Slide Time: 42:44)

Suppose a solution concept (e.g, vertex cover) is defined as a subset of objects (e.g, vertices) of the instance.

Suppose further that you have identified
a subset X of objects which
“cannot be avoided” by *any optimal solution*;
specifically,
every optimal solution has a non-trivial intersection with X .

Now, let me just conclude here by talking about a very common branching strategy. So, let us say that you are working with some sort of a subset problem like vertex cover or even feedback vertex set or hitting set any of these problems. And suppose you have managed to identify some subset of the instance, which no optimal solution can avoid. So, in some sense, every solution must intercept this subset in some way.

We do not know how, but we know that it cannot completely avoided. So, for vertex cover, this subset is simply the 2 endpoints of an edge, it could be any edge. But that is an unavoidable set for any vertex cover. For the feedback vertex said such a collection would be say, all the vertices that lie on a cycle. So, no feedback vertex can avoid the set of vertices that lie on a cycle, just by definition.

Similarly, if you looked at hitting set, then the elements of any set in the family would, you know would be a valid example for X . So, as long as, you can identify such a collection of objects that the solution cannot avoid, then the only missing piece in the puzzle is how exactly does the solution interact with this subset X ? Does it pick one of these vertices? Does it pick both of them? Does it pick all of them?

You know, depending on the example that you are working with, we do not know that. So, what the branching algorithm does for us is that it guesses this interaction, it tries all possibilities and it comes back to you with the one that worked out best. So, this gives you a very generic sort of branching algorithm concept, which is that once you found such as said

X , it could be the endpoints of an edge the vertices of a cycle, a set in a family whatever it is, you can just branch into size of X many directions.

So, each one of them is an exploration of the possibility that S picks up this object from this collection and that gives you like size of X to the k branching algorithm for your problem.

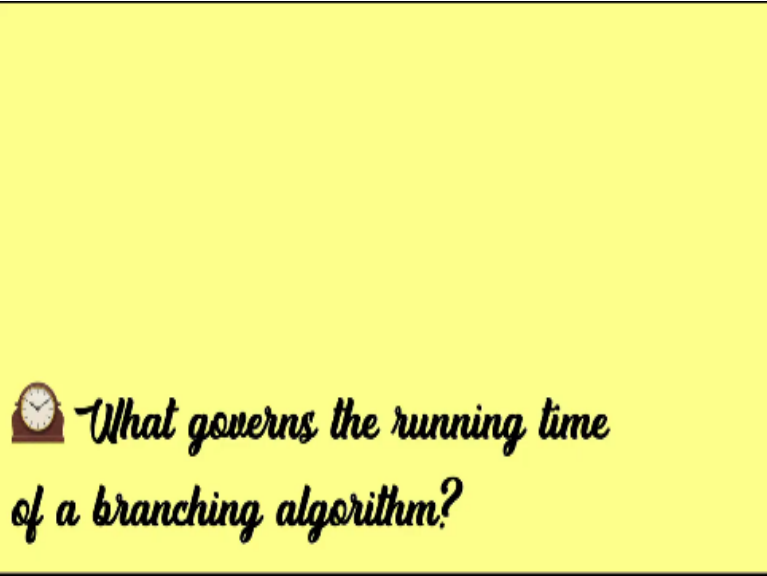
(Refer Slide Time: 45:01)


Simply branch over all possibilities in X with the measure (in this case, the solution size) dropping by one in each branch.

Once you identify the set X , you automatically have a $|X|^k$ branching algorithm with this strategy.

So, this is a sort of a general approach that gives you a first cut algorithm. But typically, what you would want to do is to use this as a starting point and build on it in some way to get to more interesting and efficient running times.

(Refer Slide Time: 45:25)



 *What governs the running time of a branching algorithm?*

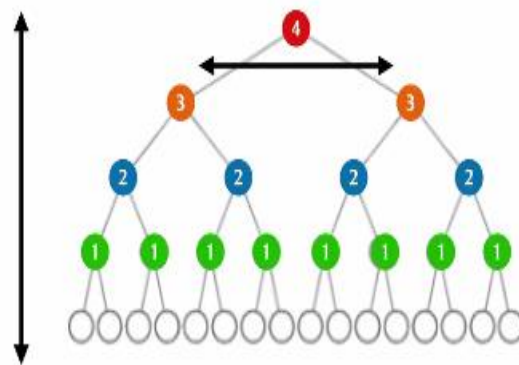
So, speaking of efficiency and running time, let me just wrap up here with a reminder of what are the main factors that contribute to the running time of a branching algorithm.

(Refer Slide Time: 45:32)

The short answer:
breadth & depth

Remember that with the branching algorithm, you typically always have a search tree that is associated with the algorithm that captures the work that algorithm is doing. And your running time is captured by both the breadth and the depth of this search tree.

(Refer Slide Time: 45:49)



So, this is a just a useful intuition to carry at the back of your mind. And keep in mind that you also may have to make some trade offs because the depth really is governed by how quickly your measure is dropping. And the breadth is, you know, governed by the number of instances that you tend to generate.

(Refer Slide Time: 46:11)

So watch out for the tradeoffs!

Do you want **100 branches** with
the **parameter dropping by 25** in each of them,
or do you want **two branches**
with the **parameter dropping by 5** in each?

What would you bet without picking up a calculator?

So, for instance, if you had to make a comparison between an algorithm that offers to get into 100 different parallel universes. But, really enjoys a very large drop of the measure in each of them versus another algorithm that generates a small number of branches, which is a good thing, but the measure does not quite drop, you know, in such an attractive manner. So, which one would you pick?

I mean, it may not be completely obvious without actually going through the calculations. And when you work more and more with branching algorithms, you will get increasingly familiar with these kinds of considerations, these kinds of tradeoffs that you might have to make. So, they are usually different things that you can do. And you will have to figure out which one is the best from the point of view of efficiency by really digging into the details and working through some calculations.

Speaking of calculations, the next thing, we will talk about is how do you deal with these linear recurrences. But that is the topic of the next video, which will just be a short overview of how you can calculate the running times of the branching algorithms that you come up with. So, I will see you there. Thanks for watching and bye for now.