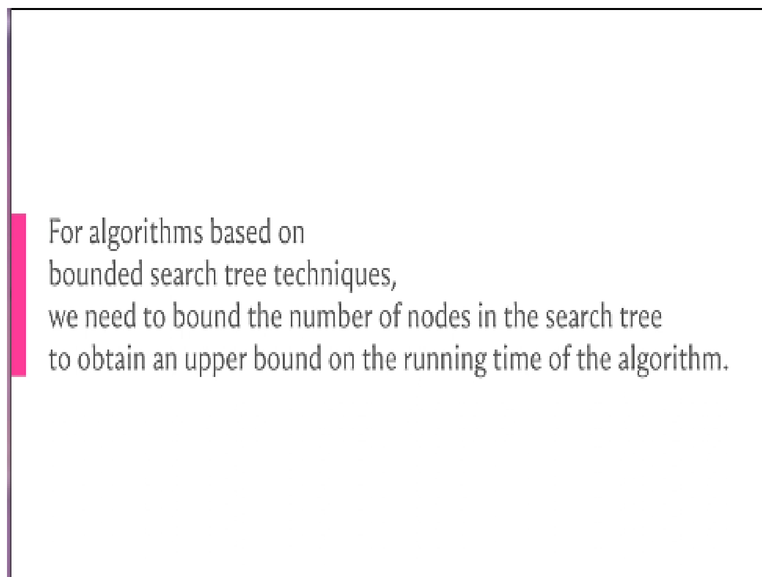
Parameterized Algorithms

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

Lecture - 08
Analyzing Recurrences

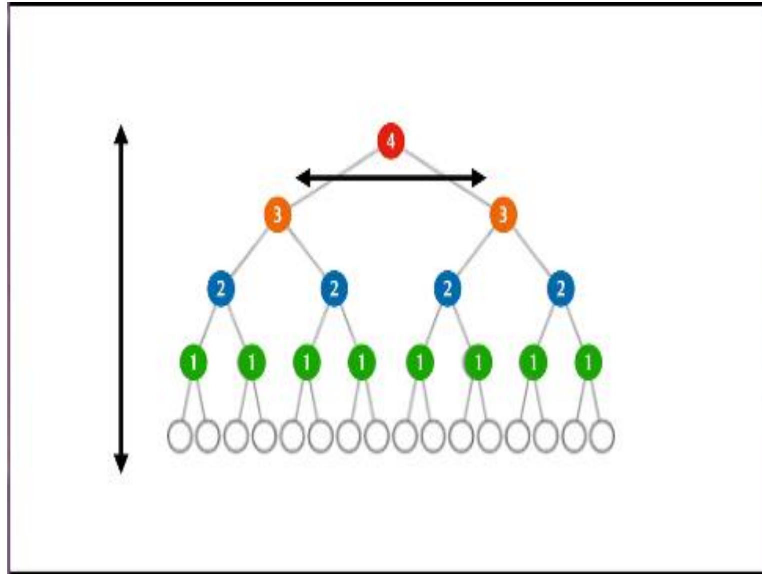
Welcome back to the second module of the second week, which as you know, by now is all about branching algorithms. So, this is a really short introduction to how you analyse the sort of references that were coming up in our previous discussion of branching algorithms. So, we will just talk about how do you handle these references? And how do you come up with the kind of bounds that we were claiming in the previous module.

(Refer Slide Time: 00:40)



So, when we are working with branching algorithms, we noticed that the work that is being done by a branching algorithm is best understood by looking at the search tree that is naturally associated with it. And we said that the work done is essentially proportional to the size of the search tree.

(Refer Slide Time: 00:55)



And by size, simply mean the total number of nodes in the search tree. But the total number of nodes is basically proportional to the number of leaves. So, we have been using the number of leaves in the search tree as a proxy for the total amount of work being done by the branching algorithm. Now, the number of leaves or at least a worst case bound on the number of leaves in a search tree is usually most naturally expressed by a reference.

And that reference is something that comes out of just studying the recursive calls that the algorithm is making, and how your concept of measure is evolving across these recursive calls. So, we have seen specific examples of these references, when we did the vertex cover branching algorithms, both when we branched on an edge, and also when we branch to vertex vertices search table.

So, I will not repeat those specific examples for you. And in fact, in the lectures that are coming up, you will see more examples of these references show up, which is why we are having this discussion right now. So, you are prepared to deal with these references as they come up.

(Refer Slide Time: 02:11)

We use recurrence relations to describe the search tree.
These are typically linear recurrences with constant coefficients.

There exists standard techniques
to bound the number of nodes in the search tree for this case.

Our treatment of this is going to be fairly short and black boxes, in the sense that is been a lot of standard techniques for handling references. And in particular, the kinds of references that we will have to worry about are mostly linear references with constant coefficients. So, these references are fairly well understood. And our plan is to take advantage of this understanding quite directly without actually opening up any of these black boxes.

(Refer Slide Time: 02:41)

If the algorithm solves a problem of size n with parameter k and calls itself recursively on problems with decreased parameters:

$$k - d_1, k - d_2, \ldots, k - d_p$$

then:

$$(d_1, d_2, \ldots, d_p)$$

is called the branching vector of this recursion.

So first, let us talk about a little bit of terminology. So, suppose you have a branching algorithm that generates p recursive sub instances. And in these p branches, the measure that you are working with, which we will just do not by k, are drops by quantities d 1, d 2, up to d p, respectively right. In this case, d 1, d 2, and so on up to d p is called the branching vector of this recursive process.

(Refer Slide Time: 03:08)

For example, in the previous lecture, the branching algorithms used:

- a branching vector (1,1) when we branched on an edge,
- a branching vector (1,2) the first time we branched on a vertex,
- a branching vector (1,3) in the final attempt.

So, just as an example, here are the branching vectors from the algorithms that we saw in the previous lecture. Hopefully, this looks familiar. So, we discussed 3 branching algorithms for vertex cover. Well, there were essentially 2 but when we branched on a vertex, we had 2 different approaches, one of which gave us a stronger bound on the number of leaves in the search tree.

So, these branching vectors are in correspondence with the references that you have already seen. And if this looks mysterious, or strange in some way, then please do pause here and go back to the algorithms that we have already discussed in the previous lecture. And try to make sure that the connection and the definition of a branching vector is clear before we move on.

(Refer Slide Time: 03:56)

For a branching vector $(d_1, d_2, \ldots, d_p)$, the bound $T(k)$ on the number of leaves in the search tree is given by the following linear recurrence:

$$T(k) = T(k - d_1) + T(k - d_2) + \cdots + T(k - d_p).$$

Assuming that to compute a new subproblem with a smaller parameter can be done in time polynomial in n, the running time of such a recursive algorithm is $T(k) \cdot n^{O(1)}$.

So, if we do have a branching vector d 1 through d p, then the recurrence for the number of leaves in the search tree for such a branching algorithm will be quite naturally given by T of k equals T of k – d 1 + T of k – d 2 and so on up to T of k – d p. That is again, just because of the form of the reference. So, the algorithm itself is generating instances where the measure has dropped by d 1, d 2 and so forth.

And so, if you draw the corresponding search trees, you will see that you have a root and you have these p sub trees that are adjacent to the root. And the number of leaves in each of these sub trees is given by the recursive expression, T of k – the appropriate drop depending on which sub tree you are analysing. And of course, for the overall tree, the total number of leaves is just the sum of all the leaves and all of these sub trees.

So, again, if you relate this back to the examples that we have discussed. In particular, when we were branching on an edge, we actually explicitly drew out the search trees. So, it may be useful for you to go back and tally that picture with the slightly more general expression that we see here. So, the overall running time is essentially given by T of k, as we have discussed before, this is with the working assumption that you only need a polynomial amount of time to generate the sub instances.

And you only need a polynomial amount of time to resolve the base cases, then you can say that the overall time that your algorithm takes is essentially governed by T of k, with this polynomial overhead. Just keep in mind that if you have an algorithm that has a different behaviour from what these assumptions stipulate, for example, maybe you have a more expensive way of handling base cases.

Or maybe you have a fancy pre processing rule, which is also expensive that needs to be run before you can generate your sub problems in a valid way. Then maybe your algorithm requires a more sophisticated analysis compared to this general approach. But I think for the most part, we are good. This certainly covers the scenarios that we will be encountering in the lectures.

So, I do not think you have to worry too much about the assumptions that we are making on the slide, but just know that these assumptions are in place.

(Refer Slide Time: 06:34)

$$T(k) = T(k-d_1) + T(k-d_2) + \cdots + T(k-d_p).$$

If we want to show that $T(k) \leq c \cdot \lambda^k$,
then the recurrence reduces to the following inequality:

$$(\lambda^{k-d_1} + \lambda^{k-d_2} + \cdots + \lambda^{k-d_p}) \leq \lambda^k$$

This can be re-written as:

$$\lambda^d - \lambda^{d-d_1} - \lambda^{d-d_2} - \cdots - \lambda^{d-d_p} \geq 0$$

So, let us take a closer look at this equation here. This, of course, is the recursive expression for T of k, which is the quantity we are interested in bounding. And with the kind of statement that we want to be able to prove is that T of k is bounded by some lambda to the k, possibly with a constant multiplier. Notice that, these were the kind of bounds that we were claiming in the previous lecture.

And notice also that we are interested in finding or discovering the smallest possible value of lambda for which this inequality holds. So, right now, we have no idea about what this lambda is? But let us just pretend that it is true that T of k is at most c times lambda to the k. And let us pretend that we are trying to prove this inequality by let us say, applying induction, for instance. And we want to apply induction on k.

So, let us say we know that this inequality holds for all smaller values of k. So, then to show that T of k is at most c times lambda k, what we can use is the fact that T of k – d 1, T of k – d 2 and so on after T of k – d p can be subject to the inductive hypothesis, because notice that all of the d’s are strictly greater than zero. So, each of these quantities, the k – d is strictly smaller than k.

So, we can see that T of k – d 1, for instance, is at most c times lambda to the k – d 1, and so on and so forth. So, let us go ahead and make those substitutions. And so, we know that this expression that you are seeing here on the left of the inequality at the bottom of your screen. We know that that is exactly T of k that is just by the definition above and the application of the inductive hypothesis, as I described it right now.

So, this expression is what we want, as being at most c times lambda k. So, I have implicitly cancelled out the c's here, there is every term in this expression could have the c multiplier, but you will see that they all cancel out. So, if you rewrite this inequality, you essentially get this sort of equation or rather, this inequality, which you can view as being a polynomial in lambda. And by the way, what is d here? So, d is the largest of the d's.

So, essentially, we are cancelling off whatever we can, and this is what we will be left with. So, we have this polynomial in lambda, which we want to be non-negative. And we want to know, what is the smallest value of lambda for which this inequality holds? So, it turns out that the polynomial on the left hand side of this inequality has a unique positive route, let us call it lambda not.

This turns out to be the best possible choice for lambda in terms of a value that satisfies this inequality. Any value of lambda that is less than lambda not will not work, because, it violates the inequality. And since lambda not satisfies the inequality, you would anyway not be interested in any values of lambda that are greater than lambda not even if they work out. So, essentially lambda not is what we will call the branching number for this branching factor.

And that is the number for which you can say that T of k is bounded by lambda naught to the k. So, at this point, a natural question you might have is, how do you get to this lambda naught? this is really just a matter of calculation, once you have your branching factor. It is completely straightforward to write down this polynomial in lambda. And if it is a polynomial that factors easily and you can visibly identify the root, then that is great.

You may have done a similar exercise for, you know, say the Fibonacci recurrence or something like this. So, it can be a better of a manual process depending on how nasty your polynomial you are working with. One general way of doing this is to go to a website like Wolfram Alpha, actually input the polynomial and get the routes, a software like that will use basically, again, black box methods to come up with a route. And it is not something that you have to worry about if you just want to get to the number.

(Refer Slide Time: 11:10)

(i, j)	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2		1.4143	1.3248	1.2721	1.2366	1.2107
3			1.2560	1.2208	1.1939	1.1740
4				1.1893	1.1674	1.1510
5					1.1487	1.1348
6						1.1225

Table 3.1: A table of branching numbers (rounded up)

Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk and Saket Saurabh, Parameterized Algorithms, Springer

The text on parameterized algorithms actually has this handy table, if you are only working with simple branching vectors that have 2 entries in them or 2 coordinate branching vectors. And, you know, if they range from 1 to 6, then this table already gives you the values and you can simply look it up. But you might have branching vectors which have more coordinates in them.

Or maybe your drops are larger than 6, in which case, you would actually have to resort to the use of software or just maybe some clever calculation depending on the polynomial at hand. So, this was essentially an engineering perspective on how to come up with the branching number for a given branching vector. And with this background in place, we are now ready to tackle more branching algorithms.

So, in the upcoming lectures, we will be talking about the feedback vertex set and vertex cover in a new light with an exciting new parameterization. So, I will see you in those discussions. Thanks once again for watching and see you soon.