Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 01

Lecture 02

Welcome to the second lecture of the second level of algorithms. In the first lecture, we defined what amortized complexity is, and we saw how, using amortized complexity, we can build a data structure. There is a data structure called Fibonacci Heap with much better amortized time complexity than Min-Heap. And how we can use that to get a faster implementation of Dijkstra's algorithm. Later in the course, we will see Fibonacci Heap also, but we will continue now with amortized time complexity. So there are three standard methods to compute the amortized time complexity of any algorithm or operation.

The first one is called aggregate analysis, the second one is called the accounting method. The third one is called the potential function method. Okay, so let us study these methods one by one. So let us begin with aggregate analysis. This is the most direct approach for computing the amortized time complexity of an algorithm straight from the definition.

You try to compute the total time that any sequence of k operations can take in the worst case and divide it by k. That is the amortized time complexity of one operation. So compute the worst-case time complexity of any sequence of k operations. Okay, why worst case? Because there can exist multiple sequences of k operations for any integer k, and what is the maximum time any sequence of k operations can take? You divide by k, and that gives you the amortized time complexity of this operation.

So, you compute the worst-case time complexity of any sequence of k operations for any natural number k and divide by k. So, let us see an example. Example 1. Our first example is of a stack with a special operation called multipop. So, let us call it multipop stack.

So, I take any standard implementation of a stack which can push and pop in $O(1)$ time. So, if we take any linked list-based implementation of a stack, that usually takes $O(1)$ time for push and pop. So, push is insert front in the linked list, and pop is delete front in the linked list. So, the operations are as follows. So, one is push some key x. This operation pushes a new key x on the top

of the stack. Okay, next one is pop. It deletes and returns the top element of the stack, and the multipop k, some integer. So, if the stack currently

Holds s keys, then this operation returns the minimum of k and s top elements from the stack. So, return and delete. So, pop k times—that is it. So, that is multipop. If the stack has fewer than k elements, then you return all elements. If there are not enough elements to pop, then you return all the elements, and after this operation, the stack will be empty.

So, what is the worst-case time complexity of these operations? So, push, and pop and multipop worst-case So, push is $O(1)$. As I said, standard implementation—either array or linked list—takes $O(1)$ time for push and pop.

And because multipop, let us call it k, it basically pops k times. So, its worst-case time complexity is $O(k)$. Now, let us find out what the amortized time complexity is. So, this is the worst-case time complexity. And what is the amortized time complexity?

So, you will show that the amortized time complexity of all three operations is O, and for that, let us use this aggregate analysis. So, let us consider any sequence of n operations where each operation is either a push, a pop, or a multipop. So, let us consider any sequence of k operations.

So, a nice app analysis will reveal this. So, you want to bound the total time complexity of these n operations. So, we want to bound the total worst-case time complexity of the sequence of n operations. Okay, so what is the naive or straightforward approach?

Naive approach. Because there are a total of n operations, the size of the stack is at most n. at most n at any point in time in the sequence, and if that is the case What is the worst-case time complexity of any operation? Push takes O(1), pop takes O(1), and multipop takes O(n). So, the worst-case time complexity of each operation.

Hence, what is the time complexity of any sequence of n operations is bigger than n squared. Hence, the amortized time complexity of any operation is $O(\frac{n^2}{n})$, which is $O(n)$.

This is correct, but this Actually, to get an amortized time complexity of O, we do not need any technique; it just follows from the definition.

Amortized time complexity is always less than or equal to the worst-case time complexity. This follows straight from the definition; you can check it as homework. The amortized time complexity of any operation is at most its worst-case time complexity. Okay, so now we will do a tighter analysis. See, to pop an element, either through the call of pop or through the call of multipop, it needs to be pushed first.

That means every pop, either in pop or through the call of multipop, has a corresponding push operation. But the number of push operations is at most n. Hence, the number of pop operations, either through pop or through multipop, is at most and each pop takes big O of one time in the worst case. So, the worst-case time complexity of any sequence of

n operations where each operation is a push or pop or multipop is $O(n)$. Hence, the amortized time complexity push pop and multipop is $O(\frac{n}{n})$ which is $O(1)$ okay. So this is how we use aggregate time complexity or aggregate analysis to compute amortized time complexity. So let us see our second example which is incrementing a binary counter.

Incrementing a k bit binary counter that is adding one modulo 2^k Okay, so what is the worst case time complexity of this process of this operation? You see, we assume that for toggling, for changing one bit takes bigger of one time. So, changing one bit takes bigger of one time. So how is this operation performed?

We start from the least significant digit and if it is 0, we make it 1. and then the operation is done. If it is 1, I make it 0 and go to the next least significant digit and if it is 0, I make it 1 and then it is done. So, in general, I find out the rightmost the, so if I write the digit as, so this is b_{k-1} most significant digit, $b_{k-2} ... b_2 b_1 b_0$.

So, I find out in this representation the rightmost 0 and then toggle that 0 to 1 and toggle all the 1s to the right of that rightmost 0 to 0. So, find the index i of the rightmost 0. If there is no such index i that means the number is all 1 then make all bits 0. If

there is no such i then in this case the number must be all 1 and if I add 1 then because addition is modulo 2^k the result should be 0. So, if there is no such i make all bit 0. Otherwise otherwise make the i-th bit 1 and all the bits to the right of i to 0.

So, what is the worst case time complexity of this increment operation? Because in the worst case I may need to change all the k bits, the worst case time complexity of increment is $O(k)$. But in the next lecture, we will see that the amortized time complexity of increment is O and we will use again the aggregate analysis to prove this claim. So, let us stop here. Thank you.