Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 01

Lecture 01

Hello, welcome to the first lecture of the second level of Algorithms. So, our first topic is Amortized Analysis. So, till now, when we analyze algorithms, the time complexity of it, we usually use 3 metrics. One is the worst-case running time or worst-case time complexity of an algorithm, the best-case time complexity of an algorithm, and the average-case time complexity of an algorithm. So,

The commonly used performance metrics to quantify the time complexity of any algorithm, the most commonly used is the worst-case time complexity. What is it? It is the maximum number of steps the algorithm performs over all possible inputs of size n.

For example, the worst-case time complexity of insertion sort is $O(n^2)$ of insertion sort is $O(n^2)$. The second one that we also use is the average-case time complexity. So, here we assume a probability distribution over all possible inputs of size n.

So, typically most commonly used probability distribution is the uniform distribution over all possible inputs of size n, but it can be any other distribution also. Then the average case time complexity. of the algorithm is the expected number of steps the algorithm performs on an input selected using the probability distribution on the inputs.

So, for example, if we assume uniform probability distribution over inputs then the average case time complexity of quick sort is $O(n \log n)$. ok, but recall that the worst case time complexity of quick sort is $O(n^2)$. ok and the next one is the best case time complexity. it is the minimum number of steps the algorithm performs for any input of length n. For example,

the best case time complexity of insertion sort is $O(n)$. but the worst case and average case time complexity of insertion sort is $O(n \log n)$. average case is assuming uniform distribution over inputs. So, the best case time complexity of insertion sort is $O(n)$, but

both the worst case and average case time complexity of insertion sort are $O(n^2)$ time complexity of insertion sort. So, now we will study another notion of time complexity which is called amortized time complexity.

So, then what is amortized time complexity? So, here is the definition we say that sequence of we say that an operation has an amortized time complexity of $T(n)$ if any sequence of k operations execute $O(k \times T(n))$ steps for every natural number k ok.

So, let us take any sequence of k operations and the total time complexity will be $O(k \times T(n))$. So, it can happen that some intermediate operation takes more than $T(n)$ time. but for any sequence of k operations for any natural number k the total time spent will be bounded by $O(k \times T(n))$. If that is the case then we say that this particular operation has an amortized time complexity of $T(n)$. So, why do we need such a such a notion? So, let us motivate the need of amortized time complexity with a very important application. So, what is the need of amortized time complexity? So, for that let us recall the single source shortest path problem and the Dijkstra's algorithm for that problem which works if all the edge weights are non negative.

So, recall distrust single source shortest path problem ah distrust algorithm for single source shortest path problem. So, if you recall the algorithm in that algorithm initially the single source is assigned a key 0 which is the distance from the source which is 0 and the every other vertices in the input graph has initialized with a key infinity and we maintain a minimum heap. So, standard implementation of Dijkstra's algorithm maintains a mean heap.

storing the vertices of the input graph ok. And in every step is an iterative algorithm in every iteration we perform an extract mean in that mean heap and the algorithm terminates when the minimum heap is empty. ok.

And in every iteration after performing the extract mean operation we update the key of the neighboring vertices which is decrease key. So, we update basically decrease we do not increase the distance value or decrease the key of vertices of the vertex which extract mean has returned. in this iteration ok.

And before entering this loop we perform a build if operation with all the vertices in the graph. So, before entering the loop we perform build heap with all vertices. the key of the source is 0 and the key of every other vertex is infinity ok.

So, now, let us analyze the running time of Dijkstra's algorithm. So, the running time is the time for building the heap plus n extract min operations, one for each vertex. Because the algorithm terminates when the heap is empty, and initially, before entering the loop, it has n keys. In every iteration, we are performing exactly one extract min. So, the size of the min heap is decreasing by exactly one in every iteration. So, n extract-min operations.

Plus, how many decrease key operations? For every vertex, we are performing decrease key operations for all its neighbors. So, if you see, for every edge, we perform decrease key at most twice. So, time for 2m decrease key operations. So, if you use a standard implementation of a min heap, then the time for building the heap is $O(n)$. Extract min can be done in $O(\log n)$ time, and we are performing n many of them. So, the total time complexity for n many extract min operations is $O(n \log n)$.

Right, and decrease key also takes $O(\log n)$ time. So, this is $O(2m \times \log n)$, okay. So, this is $O(m \log n)$. Now, there exists another data structure called Fibonacci Fibonacci heap, which performs this operation with much lower amortized cost. So, these are the operations: build heap, extract min.

And decrease key, OK. The amortized cost of these operations is as follows. Build heap takes $O(n)$ time, extract min takes $O(\log n)$, and decrease key takes $O(1)$. These are not worst-case bounds; these are amortized bounds. But now you see, any amortized guarantee says that any sequence

So, we perform a build heap first. So, that takes $O(n)$ time, but then after that, any sequence of k_1 many extract-mins and k_2 many decrease keys takes time $O(k_1 \times \log n) + O(k_2)$. That is the meaning of this line. So, this means that any sequence of k_1 extract-mins and k_2 decrease keys takes $O(k_1 \times \log n) + O(k_2)$ time.

Now, you see, if I use Fibonacci heap instead of min heap in the implementation of Dijkstra, then the runtime of Dijkstra using Fibonacci heap is $O(n)$ for build heap, then we are performing n many extract-mins. So, n log n then we are performing m many decrease keys, 2m many decrease keys. So, big O of m. So, the runtime becomes $O(m + n \log n)$, which is much faster than the implementation of Dijkstra's algorithm using min heap.

So, with this motivation, in the next lecture, we will see various examples of how to compute the amortized time complexity of various operations and algorithms, okay? So, let us stop here. Thank you.