Course Code: CSE 215

Course Title: Algorithm Lab

Submitted To: Subroto Nag Pinku

Computer Science and Engineering

Daffodil International University.

Submitted By:

ID	Name	Email
191-15-12971	Koushik nag Shuvo	Koushik15-12971@diu.edu.bd
191-15-12995	MD. Sakib Uddin	sakib15-12995@diu.edu.bd
191-15-12971	Asadullah Al Imran	Asadullah15-12981@diu.edu.bd
191-15-12979	Joy Sarkar	joy15-12979@diu.edu.bd

To understand which is beater algorithm, we have to measure the Mathematical and logical sequence of an algorithm. The two important factor which decide the efficiency of algorithm is,

- Time complexity
- Space complexity

We can compare two algorithms by providing large amount of input, and by checking which one provides result faster, then we decide which one is beater.

So, finding the beater we always have to find out the best case. and to find the best case we have to calculate the order of an algorithm. The notation used to describe the order is the O-notation. In this notation an algorithm in which the primary logic is executed N^2 times for a problem of size N is said to have order N^2, or $O(N^2)$.

Let me give an example: Say that you have an array of strings, in no particular order, and you want to find a specific string in the array. Since you don't know where the string is in the array, you will have to perform a linear search from left to right (index 0 until the largest assigned index) to see if the index you are currently looking at refers to a slot in the array containing the string that you are looking for. In the best case, the string is the first element (you never have to traverse) in the array. In most cases, it's probably somewhere in the middle. But what we care about for the purpose of calculating the Big O (just another phrase for the "order") is the worst case. So let's consider how we could get worst case scenarios for searching for a string in an array: Case 1: The string is the last element in the array. So we had to traverse the entire array to find the string. Case 2: The string is not even in the array. So we had to traverse the whole array, just to find out that the string was in fact, not in the array. Both Case 1 and Case 2 share something in common: we had to run our check step (where we see if we've found the string) n times for an array of n items. So, we represent a linear search algorithm's Big O (or complexity, just another term for Big O or order) as: O(n) This means: a linear search algorithm performs on the order of n steps. This notation and how you say it (the "Big O" or "complexity") is so common in programming parlance that you will likely encounter it in programming interviews. The interview will say something like "Come up with an algorithm to solve this problem, then let's mock it up with some code, and afterward tell me your algorithm's complexity or Big O". For this specific problem (finding an element in an unsorted array), we can't make an algorithm any faster than linear search to find the string we're looking for because we have no idea where it is in the array. But let's change the problem slightly: say that the strings in the array are now sorted.

Koushik nag Shuvo _191-15-12971

“Better” is kind of vague. There is a way of estimating in general how slow or fast an algorithm is, but it’s mostly for comparison against other algorithms and how slow or fast they are. What you do is calculate the “order” of the algorithm. This is shorthand for “the algorithm will complete in the order of some polynomial number of steps”. The notation for this is: O(some polynomial dependent on input size n) That’s a capital O, open parent, some polynomial (like n^2, if the input size is n), and a close paren. The way in which you calculate this order is by making worst case assumptions. Let me give an example: Say that you have an array of strings, in no particular order, and you want to find a specific string in the array. Since you don’t know where the string is in the array, you will have to perform a linear search from left to right (index 0 until the largest assigned index) to see if the index you are currently looking at refers to a slot in the array containing the string that you are looking for. In the best case, the string is the first element (you never have to traverse) in the array. In most cases, it’s probably somewhere in the middle. But what we care about for the purpose of calculating the Big O (just another phrase for the “order”) is the worst case. So, let’s consider how we could get worst case scenarios for searching for a string in an array: Case 1: The string is the last element in the array. So, we had to traverse the entire array to find the string. Case 2: The string is not even in the array. So, we had to traverse the whole array, just to find out that the string was in fact, not in the array. Both Case 1 and Case 2 share something in common: we had to run our check step (where we see if we’ve found the string) n times for an array of n items. So we represent a linear search algorithm’s Big O (or complexity, just another term for Big O or order) as: O(n) This means: a linear search algorithm performs on the order of n steps. This notation and how you say it (the “Big O” or “complexity”) is so common in programming parlance that you will likely encounter it in programming interviews. The interview will say something like “Come up with an algorithm to solve this problem, then let’s mock it up with some code, and afterward tell me your algorithm’s complexity or Big O”. For this specific problem (finding an element in an unsorted array), we can’t make an algorithm any faster than linear search to find the string we’re looking for because we have no idea where it is in the array. But let’s change the problem slightly: say that the strings in the array are now sorted. Then we could actually do better than linear search. We could do a binary search: Start in the middle of the array. Is the string in that slot the one you were looking for? If the string is less

than (lexicographically when sorted) the current string, subdivide your search region to only the left half of the current region If the string is greater than (lexicographically when sorted) the current string, subdivide your search region to only the right half of the current region Go back to first bullet point Keep repeating this pattern (continually subdividing the search region in half) until you either have the string or there is only 1 element left under consideration and it's not the one you were looking for. This algorithm is: O(log) Because there are n items in the array and we continually subdivide the search region in half, we have a logarithmic relationship (log base 2, not log base 10). So, if the array is sorted, then binary search performs better than linear search because: O(log) < O(n) "Order of log n" is less than "Order of n". This is only the first part of algorithm analysis though. The next part is actually running the algorithm to profile how well it performs with different kinds of input. But in general, you really should calculate the Big O of algorithms you come up with to solve problems because that will allow you to more formally understand how well your algorithm performs against others in a computational sense. Happy coding!

MD. Sakib Uddin_191-15-12995

There are two important factor which decide the efficiency of algorithm, Time complexity (most important) Space complexity (less priority then time complexity) Time complexity: - We can compare two algorithms by providing large amount of input, and by checking which one provides result faster. Most we concern about average case and worst case of algorithm. So, whichever algo have better average and worst time complexity, that algorithm is said to be better one. Space complexity: Now, a day we have a lot of space. But, programmed execution happens on RAM which is a costly and limited memory. So, if two programmed have same time complexity then we look for which one is taking less space during execution.

In the best case, the string is the first element (you never have to traverse) in the array. In most cases, it's probably somewhere in the middle. But what we care about for the purpose of calculating the Big O (just another phrase for the "order") is the worst case. So, let's consider how we could get worst case scenarios for searching for a string in an array: Case 1: The string is the last element in the array. So, we had to traverse the entire array to find the string.

Case 2: The string is not even in the array. So, we had to traverse the whole array, just to find out that the string was in fact, not in the array. Both Case 1 and Case 2 share something in common: we had to run our check step (where we see if we've found the string) n times for an array of n items. So, we represent a linear search algorithm's Big O (or complexity, just another term for Big O or order) as: O(n) This means: a linear search algorithm performs on the order of n steps.

Joy Sarkar_191-15-12979

The two important factor which decide the efficiency of algorithm is,

- Time complexity
- Space complexity

We can compare two algorithms by providing large amount of input, and by checking which one provides result faster, then we decide which one is beater.

The notation used to describe the order is the O-notation. In this notation an algorithm in which the primary logic is executed N^2 times for a problem of size N is said to have order N^2, or $O(N^2)$.

This is shorthand for "the algorithm will complete in the order of some polynomial number of steps". The notation for this is: O(some polynomial dependent on input size n) That's a capital O, open paren, some polynomial (like n^2, if the input size is n), and a close paren. The way in which you calculate this order is by making worst case assumptions.

Say that you have an array of strings, in no particular order, and you want to find a specific string in the array. Since you don't know where the string is in the array, you will have to perform a linear search from left to right (index 0 until the largest assigned index) to see if the index you are currently looking at refers to a slot in the array containing the string that you are looking for.

But what we care about for the purpose of calculating the Big O (just another phrase for the "order") is the worst case. So let's consider how we could get worst case scenarios for searching for a string in an array: Case 1: The string is the last element in the array. So we had to traverse the entire array to find the string. Case 2: The string is not even in the array. So we had to traverse the whole array, just to find out that the string was in fact, not in the array. Both Case 1 and Case 2

share something in common: we had to run our check step (where we see if we've found the string) n times for an array of n items. So, we represent a linear search algorithm's Big O (or complexity, just another term for Big O or order) as: O(n) This means: a linear search algorithm performs on the order of n steps.

Asadullah Al Imran_191-15-12971