Lecture 3: Computational Complexity
Modeling Social Data, Spring 2019
Columbia University

Kiran Ramesh

February 11, 2019

1 Computational Tractability

Computational tractability is used to define whether an algorithm is practical or not with the current state of the art in computing. We need to trace how long an algorithm takes to run in terms of the input size (Note that input size does not always refer to number of inputs). Consider brute force cracking a password containing only 0s and 1s. This takes worst case 2^n tries where n is the length of the password. T(n) by convention is denotes the running time of an algorithm with input size n. This type of asymptotic analysis helps in a machine independent way of comparing algorithms based on the input size. A key assumption made is that comparison, indexing, arithmetic operations take constant time. A few general orders of T(n) are listed below :-

1. Exponential T(n) = k^n where k is a constant
2. Polynomial T(n) = cn^d where c, d are constants
3. Linear T(n) = cn where c is a constant
4. Logarithmic T(n) = logn
5. Constant T(n) = c where c is a constant

Note that logarithmic time algorithms will not look at all the data points even once. And constant time algorithms take the same time irrespective of the size of the input.

Note that there are a class of algorithms called pseudo-polynomial algorithm which does not depend on the number of inputs but rather the value of the input. Complex examples are the 0-1 Knapsack, Subset Sum problems. A more simple example is illustrated below. Here the algorithm has only one input. But the time the algorithm takes depends on the value of the input.

```
1 def gotToZero(sum):
2     while sum > 0:
3         sum -= 1
```

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure 1: Table comparing input size with computation time

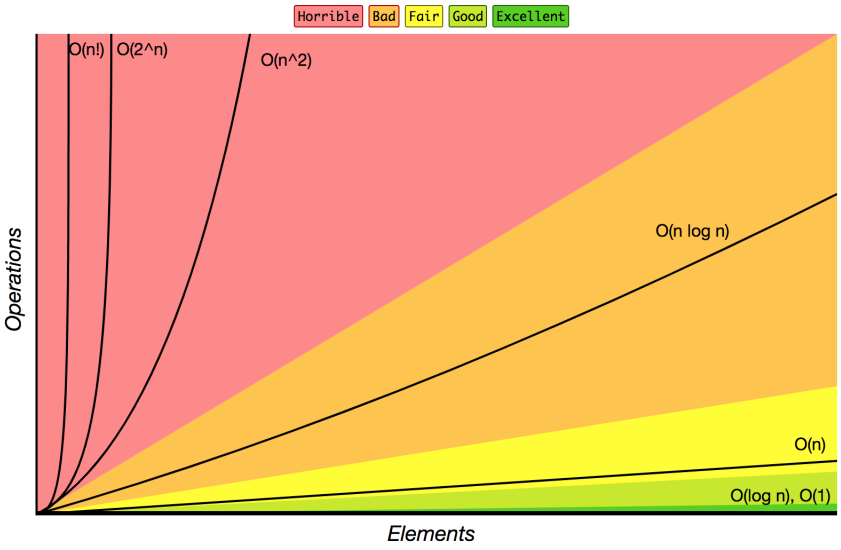

Figure 2: Graph depicting orders of growth Reference Link

2 Analysis of Runtime

Note that the algorithm will not the same time for all types of inputs even if the values are same.

1. Worst Case : In the worst case analysis, we calculate upper bound on running time of an algorithm which stands for that input which causes the maximum number of operations to be executed. For example, in the case of linear search, the worst case input is when the element we are searching for is in the last position. So worst case T(n) = n.

2. Average Case : In average case analysis, take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. For the simple linear search example, the various inputs are the inputs where the key element to be searched are in the different places in the array. Average Case T(n) = $\frac{1}{n}\sum_{i=1}^{n} i$.

3. Best Case : In best case analysis, we try to find the input which causes the minimum number of operations. For the simple linear search example, the best case input happens is when the element we are searching for is in the first position. So best case T(n) = 1.

4. Amortized : Amortized analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation. Consider a problem of simple insertions into a dynamic size array. Initially size of array is 0. When an insertion occurs, overflow happens, array size is expanded to 1. When another insertion occurs, overflow happens, array size is doubled to 2 and so on. The algorithm and analysis is explained in Figure 2 and Figure 3. Reference : Link

Note that typically worst case and average case of algorithms are taken into consideration and not the best case because there is no practical use of best case analysis.

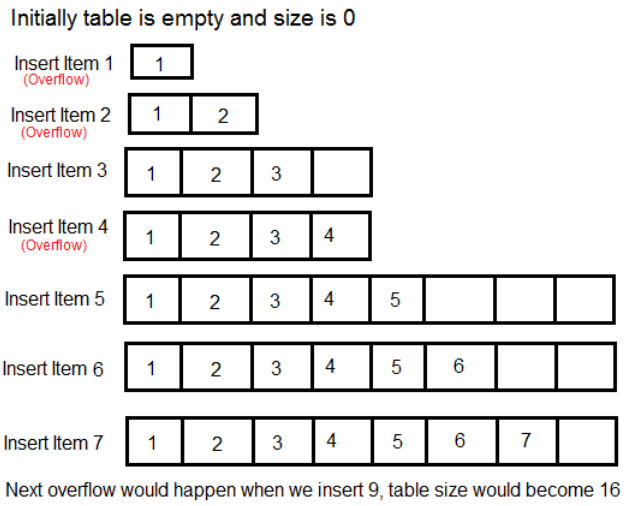

Figure 3: Example of dynamic table

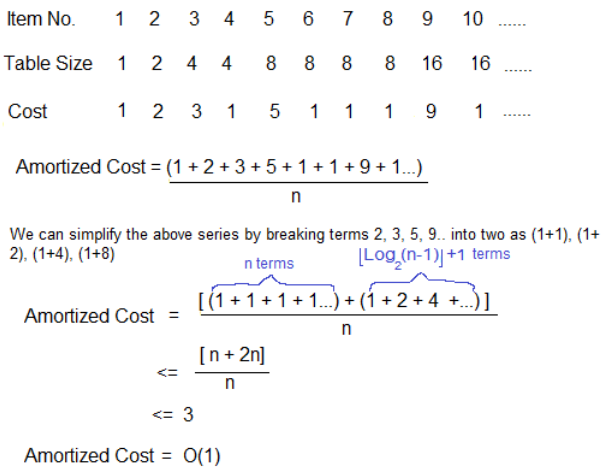

Figure 4: Dynamic table complexity analyis

3 Orders of Growth

For easier comparison of algorithms, we approximate T(n) by orders of growth. We consider only the dominant term of T(n), since the lower-order terms are relatively insignificant for large values of n. We also ignore the dominant term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. An algorithm is typically "considered" to be more time efficient than another if its worst case running time has a lower order of growth. However, note that due to constant factors and lower order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth.

3.1 Big-Oh : Upper Bound

O(g(n)) = f(n) $|\exists$ c > 0 and n_0 such that $\forall$ n $\geq n_0 : 0 \leq f(n) \leq cg(n)$

Example :-

$$T(n) = 32n^2 + 17n + 1$$
$$T(n) = O(n^3)$$

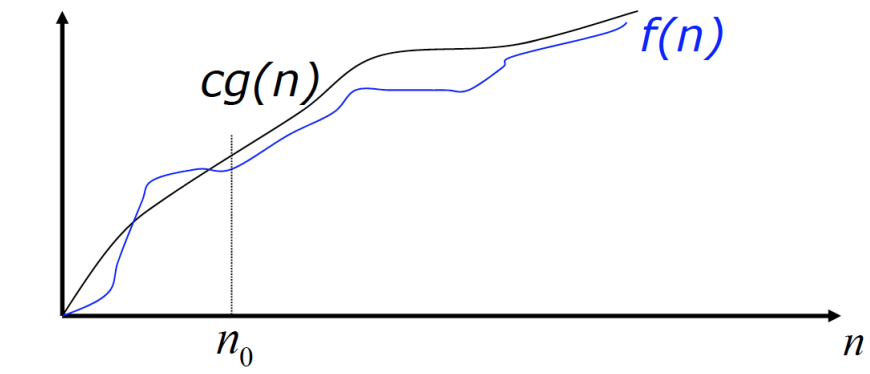

Figure 5: Big-Oh graph

3.2 Big-Omega : Lower Bound

Ω(g(n)) = f(n) $|\exists$ c > 0 and n_0 such that $\forall$ n $\geq n_0 : 0 \leq f(n) \leq cg(n)$

Example :-

$$T(n) = 32n^2 + 17n + 1$$
$$T(n) = O(n)$$

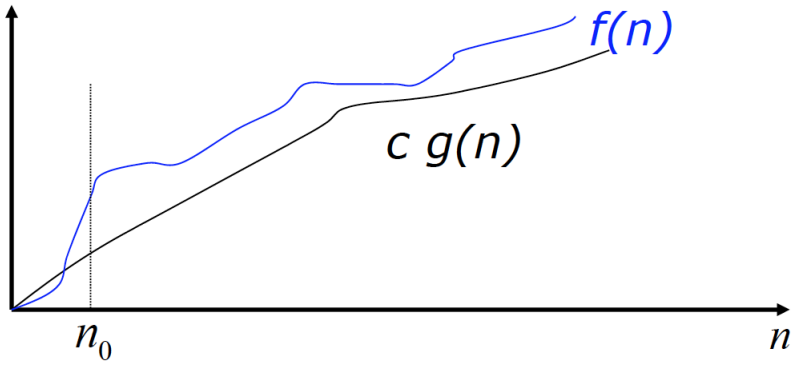

Figure 6: Big-Omega graph

3.3 Big-Theta : Tight Bound

Θ(g(n)) = f(n) $|\exists$ c1, c2 > 0 and n_0 such that $\forall$ n $\geq n_0 : c1g(n) \leq f(n) \leq c2g(n)$

Example :-

$$T(n) = 5n^2 + 17n + 1$$

$$T(n) = \Theta(n^2)$$

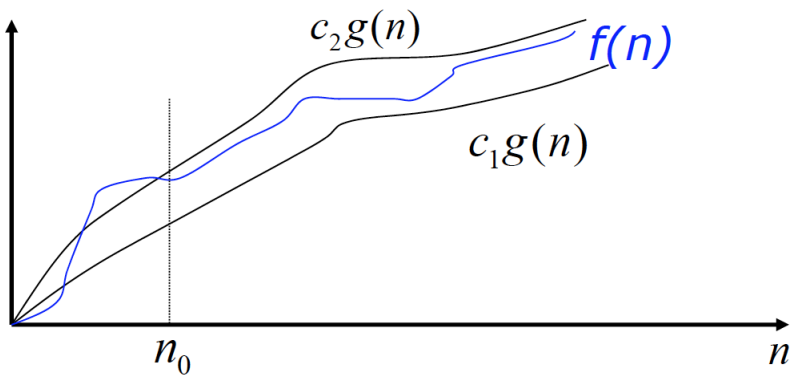

Figure 7: Big-Theta graph

3.4 Little-Oh

o(g(n)) = f(n) $|\forall$ c > 0 $\exists$ n_0 such that $\forall$ n $\geq n_0 : 0 \leq f(n) \leq cg(n)$ f(n) = o(g(n)) means that for large n, function f is smaller than any constant fraction of g

Example :-

$$T(n) = 5n$$
$$T(n) = o(n^2)$$
$$T(n) \neq o(n)$$

3.5 Little-Omega

ω(g(n)) = f(n) $|\forall$ c > 0 $\exists$ n_0 such that $\forall$ n $\geq n_0 : 0 \leq cg(n) \leq f(n)$ f(n) = ω(g(n)) means that for large n, function f is larger than any constant multiple of g

Example :-

$$T(n) = 5n^2$$
$$T(n) = \omega(n)$$
$$T(n) = \omega(n^2)$$

Notation	Ratio *f(n)/g(n)* for large *n*
$f(n) = \omega(g(n))$	$f(n)/g(n) \rightarrow \infty$
$f(n) = \Omega(g(n))$	$c \leq f(n)/g(n)$
$f(n) = \Theta(g(n))$	$c_1 \leq f(n)/g(n) \leq c_2$
$f(n) = O(g(n))$	$f(n)/g(n) \leq c$
$f(n) = o(g(n))$	$f(n)/g(n) \rightarrow 0$

Figure 8: Summary of asymptotic analysis

3.6 A few rules of thumb

1.
$$T(n) = a_0 + a_1 n + ... + a_d n^d$$
$$T(n) = O(n^d)$$
$$T(n) = O(n^d)$$
$$T(n) = O(n^d)$$

2.
$$T(n) = log_a(n)$$
$$T(n) = O(\frac{log_b(n)}{log_b(a)})$$
$$T(n) = O(log_b(a)) \ \forall \ a, b > 0$$

3.
$$T(n) = n^d$$
$$T(n) = O(r^n) \ \forall \ r > 1, d > 0$$

4 Common Running Times

4.1 Linear Time

Examples : Finding an element (Linear Search), Finding minimum, Finding maximum

Note that insertion into a linked list is not a linear time algorithm because in addition to maintaining a head, we also maintain a tail and insert values at the end and hence a constant time algorithm.

Insertion into a linked list at a specific position need not be a linear time because we can maintain a hash table where key is the index and value is the pointer to the node at index/position in the linked list. Now insertion will take O(1) time. However, this takes O(n) space. Designing algorithms is typically a trade-off between space complexity and time complexity.

4.2 Logarithmic Time

Examples : Balanced Binary Search Tree - Search and Insert. Binary Search on a sorted array.

In binary search on a sorted array, we look at the middle element and compare it to the element we are searching for, and hence reduce search space by half for each comparison we make. Therefore in the worst case we take logn time to find the element.

Skip lists is a probabilistic data structure built on top of linked lists with O(log(n)) search and O(n) space. Refer here for more information.

4.3 Linearithmic Time - O(nlogn)

Examples : Quick Sort Average Case

In quick sort, the algorithm picks a pivot element and partitions the array based on the pivot so that elements lesser than the pivot come to the left of the pivot and elements greater then the pivot to the right. This is done recursively on the left and the right till the whole array is sorted. So in the best case, the pivot chosen will always be the middle position elements of the subarray. This leads to O(n) work being done on each level while the array size gets halved every level. Therefore, O(nlogn) time complexity in the best case. Average case (using percentile analysis) is the 1/4 and 3/4 partition which gives again O(nlogn) time complexity.

4.4 Polynomial - O(n^2)

Examples : Bubble Sort, Quick Sort Worst Case

In bubble sort, we do all pairs comparison. That is the algorithm takes approx O(n) per element. Since there are n elements, bubble sort takes O(n^2) time.

In the worst case of quicksort, the pivot picked is always the greatest of smallest element of the subarray. This gives us the recurrence T(n) = O(n) + T(n-1) which gives us T(n) = O(n^2).

5 Applications

5.1 Lower Bound

An application of lower bound analysis is during algorithm design when there is a formal proof that you can do no better than then lower bound during optimization. An example is the formal proof outlined in this paper for comparison based sorting algorithms to have a lower bound of nlogn for average case.

5.2 Join

Consider that we have two tables T1 and T2 with columns id, A, B and id, C, D respectively. Supposed if we wanted to join theses two tables to get id, A, B, C, D. One way to accomplish this would be for every element in A run through every element in B to find the rows of B with matching ids. This would take $O(n^2)$ time. A better approach would be to sort the table T2 and for each id in T1 do a binary serach to find the first occurance of id. This would take approximate O(nlogn) for sorting T2 and O(nlogn) for the joining process which results in O(nlogn). The idea is illustrated here. Another approach would be to iterate through T1 and put all the rows into a hash table indexed by the id. Now for each row in T2, just refer to the hash table by id to get the rows of T1 with the same id. This takes O(n) time. A detailed explanation is provided here.

5.3 Universal Data Structure

Consider a wordload with inserts followed by lookups followed by range queries. The task is to come up with a data structure that is best suited for the workload. From earlier analysis, linked list would be the best for the insert phase with O(1) time. For the lookup phase, hashmaps would be ideal with O(1) lookup. For range queries, trees would give good performance. However, there is no known data structure that would perform best in all three phases. There is research going on to design a data structure that would perform good in all three phases maybe using morphing.