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

February 8, 2019

Notes from kkn2112

1 Introduction

In this guest lecture by Siddharth Sen, he talks about computational complexity along with exploring some data structures.

2 Runtime of algorithms

Frist, we talk about the run time of algorithms.
The run time is decided based on the number of steps executed. The more steps an algorithm needs, the longer it will take.
Popular run times are:

2.1 Exponential

Example: 2^n
This run time is usually there for brute force algorithms. Even for a small enough input size, these algorithms take a lot of time to run.

2.2 Polynomial

Polynomial run times are of the form n^d where n is the input size.
We usually consider a polynomial time algorithm efficient enough in theory and hence theoreticians often stop once they have a polynomial time algorithm.
The are generally in the form of cn^d where $c, d > 0$.
The constants c and d are important here since their scale tells us whether it is a practical run time or not.

2.3 Linear

(This is a special case of polynomial time)
Linear time algorithms take cn number of steps to reach completion, where c ¿ 0 .
An example of linear algorithms is linear search.

2.4 Logarithmic

Logarithmic runtimes are of the form clogn. The algorithms that take logarithmic time include binary search.

2.5 Constant Time

Certain operations such as addition or popping from a stack are considered to take constant time. Since they involve a fix number of operations irrespective the input size.
Sublinear algorithms are the ones where we think "how do we work on data without looking at every data point"
Usually these involve preprocessing which includes doing a lot of work upfront and then spaced out pre-processings

3 Analysis of Algorithms

We shall now see various types of analysis that help bind the run time and give us a good estimate of how long it takes for the algorithm to run.
1. Worst Case
2. Average Case
3. Best Case

3.1 Worst Case:

What if the input is worst possible input
This is the one we always consider, since we want to know how well or how bad an algorithm does on a non ideal case or on the worst case.

3.2 Average case:

We ask the question how long will it take on average, what will it do in a not so bad, not so good scenario.
Many algorithms have horrible worst case time but they do really well on average cases. An example of that would be Quicksort.

3.3 Best case

Best case is when the input is in the most convenient form and we need to do the least amount of work.

There's also amortized analysis, let's look at it.
Amortized Analysis:
In certain algorithms, expensive actions are done once a while. Otherwise it is takes a small or constant time. Hence occasional operations are slow but most of them are faster.
Example: Stack's push and popall operations
Doubling technique for dynamically sizing arrays is an example of such an algorithm.

4 Orders of Growth: Asymptotic Complexity

How do we do analysis of best case worst case and average case? By using orders of growth.
An order of growth describes how a function's runtime grows as the size of the input grows. We calculate orders of growth using three notions:

4.1 Upper Bound:

This says that the runtime of the algorithm won't grow faster than the function f.
T(n) = O(f(n))
To find the runtime in functions, we look at the dominant term from the time complexity analysis.
We often use only simple functions, the terms other than the dominant term dont have to be included.
Example: for T(n) = O(n^4) + O(n^2) O(n), We can safely ignore n^2 and n.
T(n) = O(n^4)

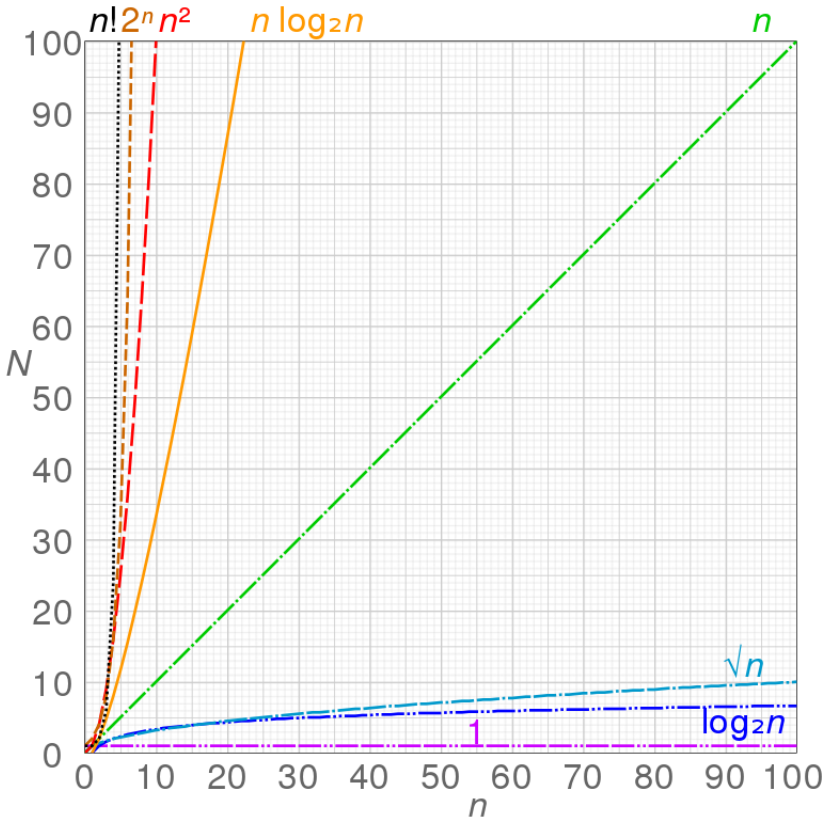
4.2 Lower Bound:

This says that the runtime of the algorithm will grow at least as fast as this function.
T(n)=Ω(f(n))
Our algorithm will always take more time than function f(n)
We can make our thing lower bounded by a lot of things but it is useful to find the one that is the closest since a vague idea of what it grows faster then is not very helpful. We hence need a tight characterisation of how long it will take.

4.3 Tight Bound:

Theta $\Theta(f(n))$:
T(n)=O(f(n))= $\Omega(f(n))$
T(n)=Θ(f(n))
In this case, the functions for lower bound and upper bound are both the same. The constants are the only things

that differ. f(n) sits in between two constants.



If we want to compare two functions to see which one grows faster, we can use limits. Take the limits of the ratios as n $\rightarrow$ infinity. One will dominate.

5 Rules of thumb:

These are some rules of thumb regarding order of growth.
1. T(n)=a + an + an^2 + an^d = O(n^d)
Only the largest term matters
2. O($log_a n$) = O($log_b n$)
The base of log doesn't matter, they would both grow simultaneously.
3. $N^d = O(r^n)$
Any polynomial run time will be dominated by exponential function eventually.

6 Common Run times

Now let's look at some common running times:
1. O(n): Finding max, min in a list, linear search, sum etc.
Any algorithm that needs to visit every data point once would have a linear runtime.
2. O(logn): Finding an item in a tree, binary search
Any algorithm where the size of problem is cut into half and one part is worked upon while the rest are discarded

would take logn time.
3. O(n^2) :Bubble Sort, pairwise comparisons, quicksort
4. O(n^3): Count the number of triangles in a graph
5. O(nlogn): Divide and Conquer algorithms

An interesting data structure:
Skip List: Linear structure which gives logarithmic insertion time. It has logn search time and logn insertion time. More on it here[1]
Linked Lists are a lot easier to update, binary trees involve a lot of moving things around.

7 Joining two tables

A naive algorithm to join two tables is as follows:
Given the IDs are unsorted, get id x, and search fpr that id through table b. Search through all entries and stop when you find the id. This takes O(n^2) time. If we remove the id that we're done with, we get an arithmetic series which gets capped to $O(n^2)$. To further cut down the time, we could sort them on their ids. And then given an id, do a binary search on the second table. This takes nlogn time to join.
Tables already have indexes in SQL, they help make things like search and join faster.
Can we go faster than nlogn?
Turns out we could if we use hashtables. Using hashtable we could bring it to order n.
Since hashing takes constant time, we dont need to worry about the time taken to produce a key from an input. We only have to produce a key and if the key causes a collision, combine and produce a new row.
This would mean we go through an id in bith rows only once, hence making it a linear time algorithm.

8 Hashtables

Hashtable maps keys to values. Each value is converted into a key which stores the value of the element.
The hashtable is usually much larger than the expected number of keys in order to deal with collision.
Collision happens when two inputs are mapped to the same key
Hashtables have constant time insertion.

1: https://en.wikipedia.org/wiki/Skip_list

Notes from kr2789

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
```

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

	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

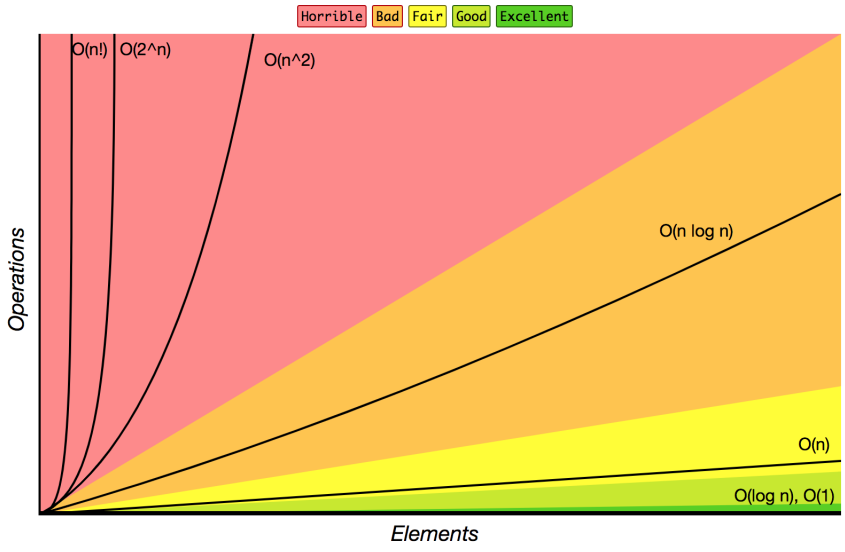


Figure 2: Graph depicting orders of growth Reference Link

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.

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 :-

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

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 :-

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

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 :-

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

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 :-

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

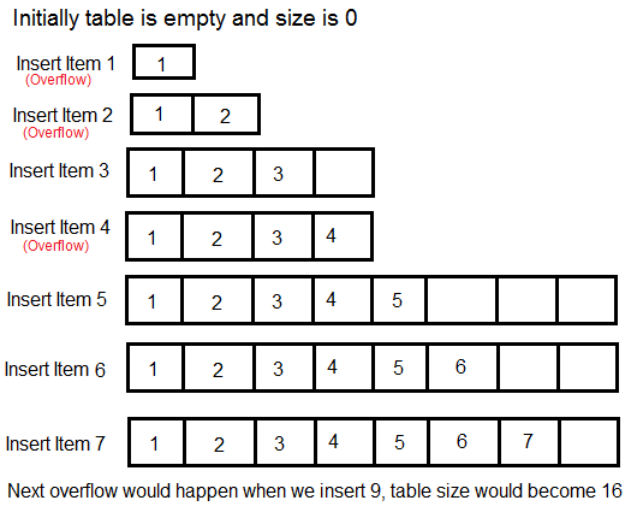


Figure 3: Example of dynamic table

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

Figure 4: Dynamic table complexity analyis

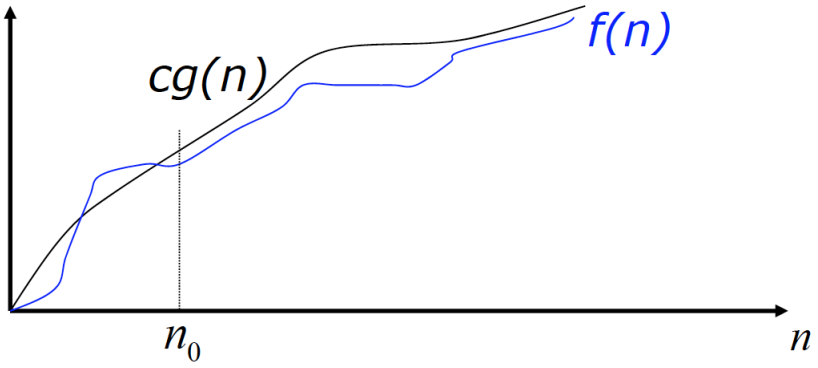


Figure 5: Big-Oh graph

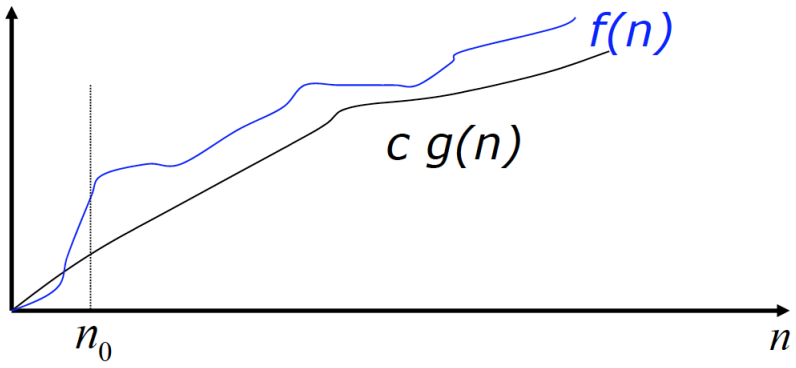


Figure 6: Big-Omega graph

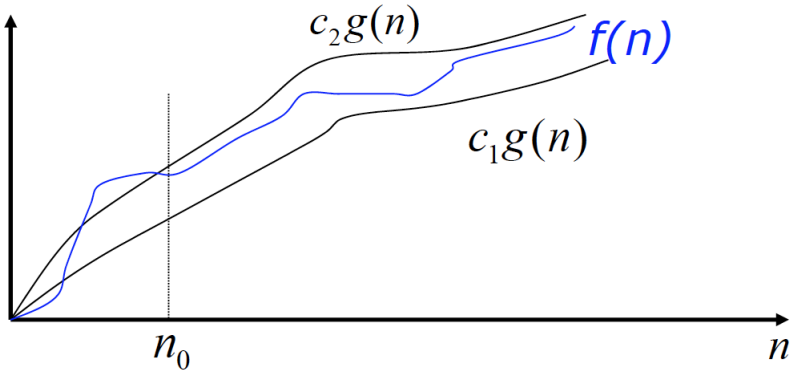


Figure 7: Big-Theta graph

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 :-

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

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.

Notes from nkt2111

1 Part 1: Guest Lecture

The first part of class was a guest lecture by Sidharth Sen, who works with Jake at Microsoft Research here in NYC. Sid's research focuses on algorithms and systems and networking and Jake noted that one of his more notable research topics involved weak AVL trees. Sid's background has a large data structures influence, and his lecture focused on exploring the question of how to store and manipulate data and how long things take.

Sid's aim for the lecture was to explore the following four topics:

1. Computational tractability
2. Asymptotic order of growth
3. Common running times (data structures, algorithms)
4. Applications

1.1 Computational Tractability

We started by roughly defining computational tractability as how to tell how long things take. Another term associated with this idea is running time, T(n), which we take to be a function of the input size n. Sid highlighted some examples of common running times. Here, it is important to note that in general, constants do not matter. In other words, we will take $2n$ to be the same as n since we are most interested in asymptotic behavior. The examples are as follows:

- 2^n
  - exponential
  - not good
  - often happens with a brute force search or in a situation when we try all combinations of something
- cd^n where $c, d > 0$ are constants
  - polynomial
  - not good when we are working with big data
  - theoretical computer scientists often consider this to be ok
  - here we do have to pay attention to the constants at least a little because it is not always obvious which algorithm is worse (ie. $20n^{100}$ versus $n^{1+0.002 \log n}$)
- n
  - linear
  - good
- $\log(n)$
  - logarithmic
  - sublinear which means we don't need to look at all of the data in a set, sometimes this means that data preprocessing is involved
  - good
- c where $c > 0$ is a constant
  - constant
  - no n dependence
  - good

1.2 Asymptotic Order of Growth

Next, we moved on to looking at asymptotic order of growth, which is often expressed with big O notation. Here, Sid defined the analysis of run time in terms of:

- worst case
  - analyze how fast an algorithm runs when we have the worst possible input
- average case
  - analyze how fast an algorithm run on average when we draw input from a random distribution
- amortized
  - analyze how fast an algorithm runs when there is one big operation that happens infrequently
  - considers an algorithm's run time based on the run time and frequency of individual operations
  - Sid emphasized that it is often more optimal in many algorithms to perform an expensive operation infrequently than a slightly less expensive operation frequently
  - an example of this is when reallocating an array it is better to double the size of the array each time the array needs to be reallocated so that we can insert elements into the array frequently without needing to reallocate space (an expensive operation) and we only need to reallocate infrequently

Before diving into a more in depth analysis of big - O notation, Sid noted that in terms of speed, run times are ranked as follows: $n < n \log n < n^2 < n^3 < 1.5^n < 2^n < n!$. This is important to keep in mind when considering upper and lower bounds.

For each of these types of run time analysis (worst case, average case, and amortized), we can consider three ways of expressing the run time:

1. Upper bound

   The upper bound is defined as $T(n) = \mathcal{O}(f(n))$ if $\exists\ c > 0, n_0 > 0$ st. $T(N) \leq cf(n)\ \forall\ n > n_0$.

   ex. For example, if we have $T(n) = 32n^2 + 17n + 1$ we can say $T(n) = \mathcal{O}(n^2)$. This is evident because we could take $c = 50$ and $n_0 = 1$. It is clear that in this case $T(n) \leq 50n^2$ when $n > 1$ so $\mathcal{O}(n^2)$ is a good upper bound. It is also important to note here that we could in theory also say that $n!$ is an upper bound for this problem, but this result is not as interesting and does not give us as much information about the algorithm. Thus, it is best to select the lowest upper bound.

2. Lower bound

   The lower bound is defined as $T(n) = \Omega(f(n))$ if $\exists\ c > 0, n_0 > 0$ st. $T(N) \geq cf(n)\ \forall\ n > n_0$.

   ex. Let us again take the example $T(n) = 32n^2 + 17n + 1$. We can again take $T(n) = \Omega(n^2)$. For this case, we could take $c = 1$ and $n_0 = 1$. It is clear that $T(n) \geq n^2$ when $n > 1$ so $\Omega(n^2)$ is a good lower bound. Similarly to with the upper bound, it is best to choose the greatest lower bound. For example, saying $\Omega(1)$ does not tell us as much about the algorithm as $\Omega(n^2)$.

3. Tight bound

   The tight bound is expressed as $T(n) = \Theta(f(n))$ if $T(n) = \mathcal{O}(f(n))$ and $T(n) = \Omega(f(n))$ and $\Omega(n) = \mathcal{O}(n)$. The visual representation for this bound is given by Figure 4.

   This can further be written as: $c_1 f(n) \leq T(n) \leq c_2 f(n)\ \forall\ n \geq n_0$ and $c_1, c_2 > 0$

   ex. In the example $T(n) = 32n^2 + 17n + 1$, we can say $T(n) = \Theta(n^2)$. The upper and lower bounds are also n^2. We can take for example $c_1 = 1$ and $c_2 = 50$ and $n_0 = 1$ and can see that $n^2 \leq T(n) \leq 50n^2$ when $n > 1$.

Next, Sid highlighted some "Rules of Thumb" when considering these upper, lower, and tight bounds:

1. $T(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d = \mathcal{O}(n^d) = \Omega(n^d) = \Theta(n^d)$
2. $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$ for $a, b > 0$ (base does not matter, it is another constant)
3. $n^d = \mathcal{O}(r^n)\ \forall\ r > 1, d > 0$

1.3 Common Running Times

Next, Sid walked through some common running times and gave examples of algorithms and data structures where these running times are commonly seen. It is important to note that here, we will assume that arithmetic, indexing, and "basic operations" take $\mathcal{O}(1)$ time.

It is also essential to note something that has been absent from our discussion up to this point, but which is essential to keep in mind. That is thinking about the amount of space we take up when implementing certain data structures or algorithms. There is often a trade off between space and time, and space is important to consider because even if an algorithm operates in sublinear time, it might require storage of massive amounts of data, which could be inefficient or even impossible. Here, we will proceed with some common running times but keep the space complexity in the back of our minds as well.

1. $\mathcal{O}(\log(n))$
   Logarithmic run time is often seen when working with binary search trees, which is a commonly used data structure that has an inherent sorting structure which makes it so we do not have to visit every node. For example, inserting or searching in a BST has logarithmic run time.

   Another data structure whose associated algorithms often have this run time is a skip list. Sid noted that we can think of the skip list as a sort of multi-leveled linked list. If the bottom level has n nodes, the next level up will have $\frac{n}{2}$ nodes and the next level after that will have $\frac{n}{4}$ nodes. The pattern continues so that each subsequent layer has half as many nodes as the previous layer. This data structure makes searching more efficient because it is possible to start at the level that is farthest out, which is analogous to starting at the first/root node in the BST. Thus, the run time for searching, similar to for a BST is $\mathcal{O}(\log(n))$.

2. $\mathcal{O}(n)$
   Linear run time is often seen when we need to touch each node or element only once. This might be the run time if we are searching for one element in a list of length n.

3. $\mathcal{O}(n \log(n))$

   A commonly seen algorithm with this run time is merge sort. It is also often seen in an algorithm where we have to touch every node in a layer, but the layers are organized in a manner like that in the skip-list where each subsequent layer has half the nodes of the preceding layer. For this algorithm, each layer takes $\mathcal{O}(n)$ time and there are $\log(n)$ layers (how many times n divides in half). This leads to an overall time complexity of $\mathcal{O}(n \log(n))$.

4. $\mathcal{O}(n^2)$

   Some commonly seen algorithms with this run time are bubble sort, quick sort, or any other algorithm where all pairs comparisons are made. More generally, this run time often results when the run time is $\mathcal{O}(n)$ per operation and we have to perform each operation n times. It should be noted that the worst case run time for the quick sort algorithm is really bad if we are unlucky and start with the largest or smallest element as the pivot.

We also briefly discussed a join algorithm, which we are likely to use in this course. This is used when we have two tables, each with an id and two different columns. Let us say table 1 has an id and then columns A and B while table 2 has the same id and columns C and D. The join algorithm produces a table with the id and columns A, B, C, and D. We discussed that the least efficient way to do this would take $\mathcal{O}(n^2)$ time if we searched each id in table 2. This could be improved to $\mathcal{O}(n\log(n))$ if we implement a sort of BST structure on the ids of column 2. The best method we uncovered, however, would take $\mathcal{O}(n)$ time and involves a hash table. We did not go into depth on what a hash table is but it was mentioned that lookups in a hash table take $\mathcal{O}(1)$ time. It is important when constructing a hash table that each bin does not get too large, otherwise the benefit of quick lookups is diminished.

1.4 Applications

Next, Sid discussed how some of the topics he discussed apply to things he is actually working on in his research. He is attempting to design a "Universal Data Structure."

Problem: The problem is that each data structure has its strengths and weaknesses. In other words, a data structure that performs well for inserts might perform very poorly for lookups and vice versa. This is an issue because when working on a problem, the workflow involves numerous different tasks, as depicted in the Figure 7, and it is difficult to choose a data structure that will perform optimally for most of the tasks.

Abstract solution: Sid's goal is to design a Universal Data Structure that can shape shift to have optimal performance at different phases of the workflow. This problem has not been solved yet, and it is something that Sid often asks his potential employees about in interviews.

2 Part 2: Review of Counting and Using Command Line

We started off with some logistical notes:

- the first homework has been posted and is due on 2/21
- it was noted that in this class we do not generally write complex algorithms, rather we use libraries that have already been written but Sid's lecture is still important so that we know how to select the best library or function for a specific task and data set
- we walked through the homework
  - for number 1, we do not need to write code but we do need to be detailed and think about what resources you need for each problem
  - when a problem notes changes in the size of the data set, we should try to find the practical solution for each size rather than just using the fanciest methods for each

Next, we moved on to a review of the command line using citi bike data. Here are the commands that we explored:

- wc -l 201402-citibike-tripdata.csv
  Counts the number of lines in the data set
- -d, -f15 201402-citibike-tripdata.csv |head
  Extracts rider gender in column 15, specifying ',' as a delimiter and limits output to first 10 lines
- cut -d, -f14 201402-citibike-tripdata.csv |sort |head
  Find the earliest birth year in column 14. Exchanging "tail" for "head" would get us the latest birth year
- grep Broadway 201402-citibike-tripdata.csv |head
  Finds all trips either starting or ending on Broadway

- cut -d, -f5,9 201402-citibike-tripdata.csv |grep 'Broadway.*Broadway' |wc -l
  List all of the unique stations in column 5 that are contain Broadway by first sorting and then removing running duplicates

- cut -d, -f14 201402-citibike-tripdata.csv |grep '[0-9]' |sort |tail
  find the latest birth year in column 14, limiting to lines with a number. Here '[0-9]' means anything between 0 and 9

- cut -d, -f15 201402-citibike-tripdata.csv |sort |uniq -c
  Counts trips by gender.
  Interesting note: here, we found that more than 4 times the riders are male than female.

- We noted some particular subcommands individually. "-r" can be used to reverse sort. "-k" denotes a key, "wc" tells the number of characters, lines, and words, quotes are used in pattern matching

Next, we were briefly introduced to awk, which is extremely powerful.

- awk -F, '$5 /Broadway/ && $9 /Broadway/' 201402-citibike-tripdata.csv |wc -l
  use awk to count all trips that start and end on Broadway

- awk -F, 'counts[$15]++ END for (k in counts) print counts[k]" t" k ' 201402-citibike-tripdata.csv
  use awk to count trips by gender without having to sort. It is interesting to note here the function of counts[$15]++. This makes an array, without any initialization, and counts the number of appearances of each item in column 15

- cut -d, -f14 201402-citibike-tripdata.csv |sort |uniq -c |awk 'printf $2" t"; for (i=1; i¡=$1/100; i++) printf "*"; printf " n"'
  displays a histogram of birth years, which each * counts 100 people. This is a complicated one, and we were a bit too rushed to get all of the details

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

Figure 8: Summary of asymptotic analysis

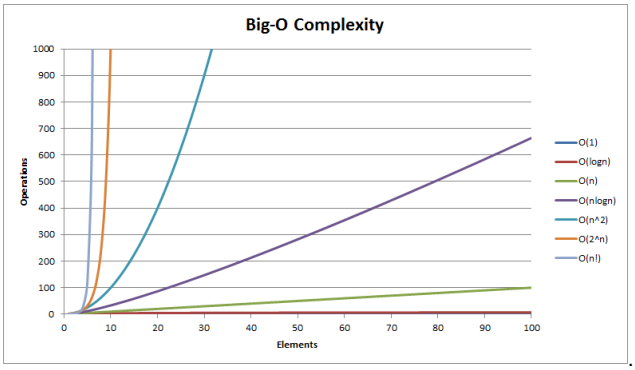


Figure 9: Run time visual comparison (from http://www.kestrelblackmore.com/blog/big-o-notation-complexity)

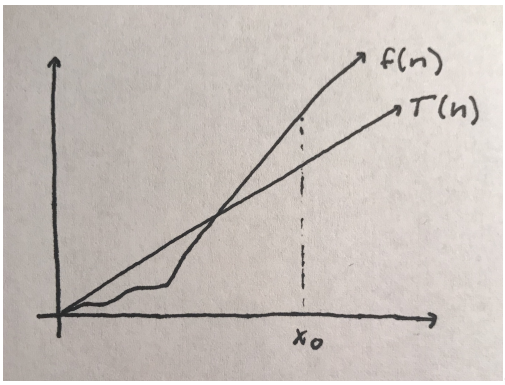


Figure 10: Upper bound visual representation

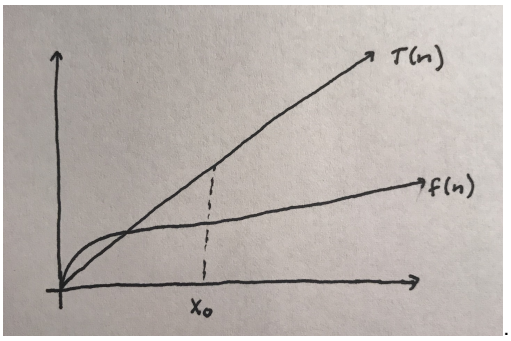


Figure 11: Lower bound visual representation

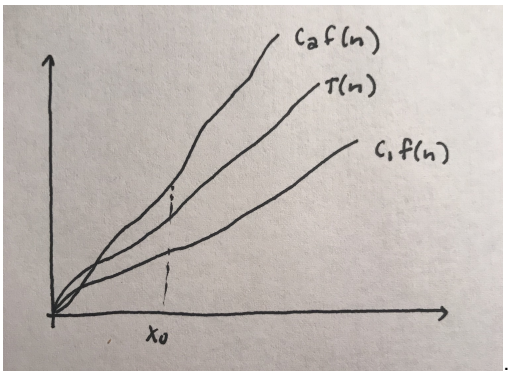


Figure 12: Tight bound visual representation

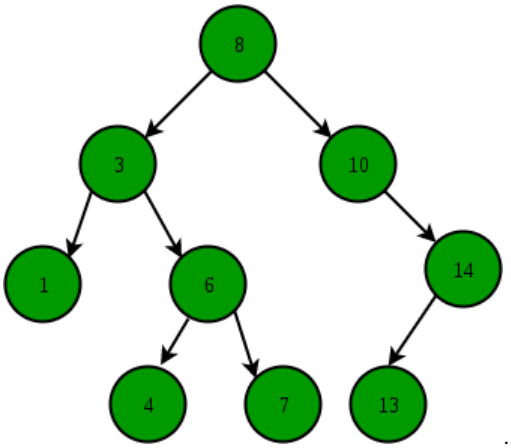


Figure 13: Binary Search Tree Visual (from https://www.geeksforgeeks.org/binary-search-tree-data-structure/)

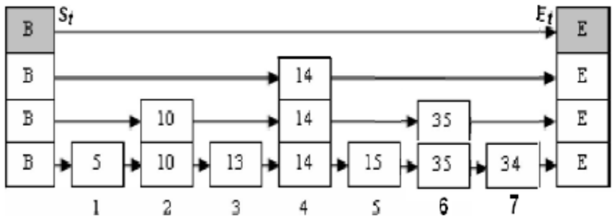


Figure 14: Skip List Visual (from https://www.researchgate.net/figure/A-skip-list-is-a-set-of-linked-lists-Numbers-within-17-are-the-indexes-of-the_fig2_221560836)

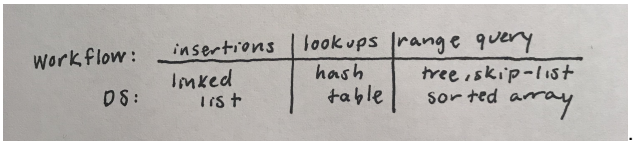


Figure 15: Workflow Diagram With Tasks and Associated Data Structures