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

Ketakee Nimavat

February 8, 2019

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:

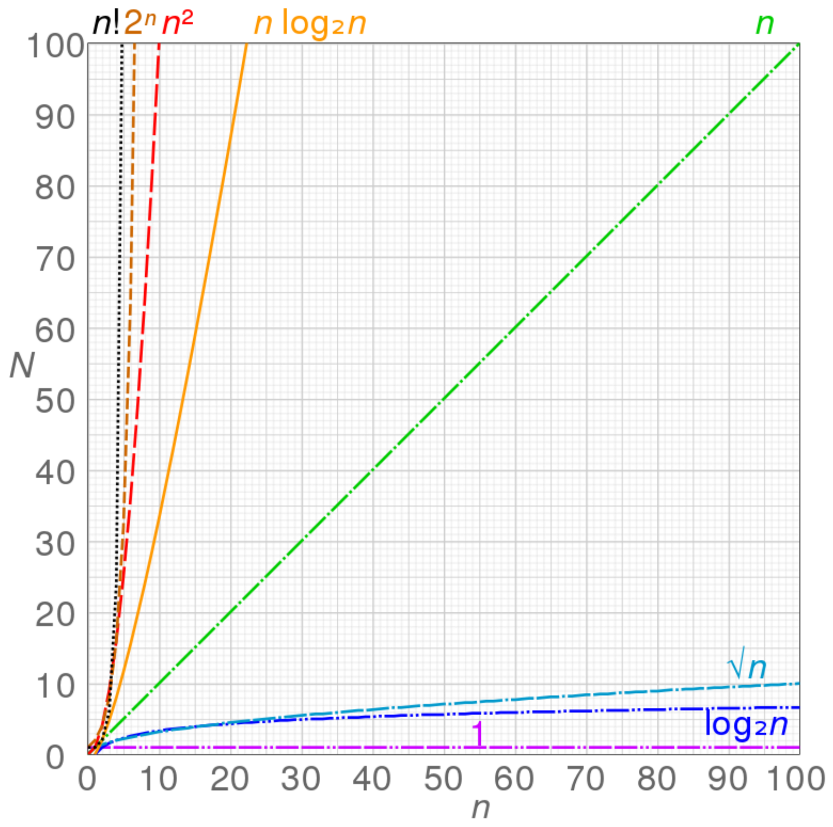
This says that the runtime of the algorithm won't grow faster than the function f.
T(n) = O(f(n))
To find the runtime in functions, we look at the dominant term from the time complexity analysis.
We often use only simple functions, the terms other than the dominant term dont have to be included.
Example: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