Lecture 3: Computational complexity
Modeling Social Data, Spring 2017
Columbia University

Kevin Zeng

February 3, 2017

1 Introduction

In determining the efficiency of an algorithm, we will want to discuss **worst case**, **average case**, and **amortized** running times, in terms of input n.

- Worst case running time is the longest an algorithm would take to run on input n.
- Average case is the amount of time an algorithm would take to run on some average input n. This requires identifying what an average input would be, e.g. what an average string might look like.
- Amortized is the amount of time an algorithm would take to run on input n making no assumptions about what average input looks like.

For example, a stack data structure has `push`, `pop`, and `pop_all` operations.

- We would say pop runs in $O(1)$ time, since we only need to perform one operation on the stack to remove one item.
- We would say `pop_all` runs in $O(n)$ time for a stack of size n, since we need to call pop once for each of n elements in the stack.

1.1 Upper Bounds

The running time $T(n)$ of an algorithm on some input of size n is $O(f(n))$ if $\exists c > 0, n_0 \geq 0$ such that $\forall n > n_0,\ T(n) \leq c_1 f(n)$.

1.2 Lower Bounds

The running time $T(n)$ of an algorithm on some input of size n is $\Omega(f(n))$ if $\exists c > 0, n_0 \geq 0$ such that $\forall n > n_0,\ T(n) \geq c_1 f(n)$.

1.3 Examples of Common Running Times

- $O(n)$ a.k.a. linear time. Example of this would be to compute the maximum number in an array of numbers.
- $O(n \log n)$. Example of this would be mergesort, average case performance of quicksort, and other divide-and-conquer algorithms.
- $O(n^2)$ a.k.a. quadratic time. Example of this would be bubble sort and worst case performance of quicksort.
- $O(n^3)$ a.k.a. cubic time. Example of this would be naive matrix multiplication.

1.4 Misc

A rule of thumb is that for every $r > 1$, $d > 0$, $n^d = O(r^n)$.

2 Hash Tables

This data structure relies on a **hash function** that maps the input to an integer between 1 and n, where n is the size of the hash table. Here are some things to know about hash tables

- Most hash tables are implemented using separate chaining, in which each element of an array contains a linked list of elements. Items are retrieved and added to the hash table by first hashing the input to determine which entry in the array it belongs to, then is appended to the linked list in that entry (if adding) or that linked list is searched (if retrieving).
- Because in most practical applications hash functions do not run in $O(1)$ time, and because of separate chaining, there is no guarantee about accessing elements in constant time.
- The hash function, if designed well, does a good job at spacing out elements in the hash table such that the load factor (average length of each linked list in the array) is minimal.
- The hash function, if designed well, has a probability of collision at $1/n$, where n is the length of the array. As a result, for n insertions, the expected number of collisions is $n * 1/n = O(1)$.

3 Join

The join operation takes as input two tables that has some data (e.g. a column) in common, and the output is one table that combines information from the two tables. For example, we can have a table where each row has a user ID and a phone number, and another table where each row has a user ID and an address. A join between these two tables on user ID would yield a new table where each row contains both the address and phone number of the corresponding user.

The following are some approaches to computing a join operation:

- For each row in A, find the corresponding row in B. This runs in $O(nm)$, where n is the number of rows in A, and m the number of rows in B.
- Create an index (e.g. using a B-tree) that spans the rows of table B. Then whenever we want to find the corresponding row in table B, it takes only $O(\log m)$ time. However, our total run time is $O(n \log m)$.
- Use a hash function on each row in both table A and B so the rows with the same ID get inserted into the same bin. Then, for each bin, perform the necessary join operations. This runs in $O(n + m)$ time, since we only go through each element in both tables twice: once to put them in the corresponding bin, and once to combine it with the other element in the bin.

4 Intro to R

The rest of the class was spent on slides and demonstrations on RStudio.