CSE548 & AMS542: Analysis of Algorithms, Fall 2017 **Lecturer:** Rezaul A. Chowdhury
Topic: Introduction **Date:** 28th August, 2017
Scribe: Bhavesh Goyal

1.1 Some useless Information about the Course

Lecture Timings: MW 7:00 pm - 8:20 pm

Location: Javits Lecture Hall 102, West Campus

Instructor: Rezaul A. Chowdhury

Office Hours: MW 5:00 pm - 6:30 pm @239 Computer Science

Email: rezaul@cs.stonybrook.edu

Teaching Assistant(s): Not yet decided as per when the lecture was scribed

ClassWebpage: http://www3.cs.stonybrook.edu/ rezaul/CSE548-F17.html

(All slides would be posted on the course webpage)

1.2 Course Prerequisites

- Fundamental Knowledge of undergrad level Data Structures(includes but not limited to Stacks, Lists, Arrays, Queues)

- Discrete Mathematical Structures (includes but not limited to Graphs, Trees and their adjacency matrix and list representations)

- Basic programming techniques (Recurrence, Sorting, Searching, Divide and Conquer, Dynamic programming, Asymptotic Analysis)

- Basic knowledge of asymptotic notations ($\mathcal{O}(.)$, $\Omega(.)$ and $\theta(.)$ concepts)

1.3 Tentative Syllabus

- Recurrence relations and divide-and-conquer algorithms

- Dynamic Programming

- Graph Algorithms (e.g., network flow)

- Amortized analysis

- Advanced data structures (e.g., Fibonacci heaps)

- Cache-efficient and external-memory algorithms

- High probability bounds and randomized algorithms:

For large enough data sets, even $\mathcal{O}(n)$ complexity is not acceptable at times because of space and time

limitations. So we try to devise approximate techniques and try to get the results correct **most** of the times.

- Parallel algorithms and multithreaded computations

Are these really important? Yes. It is dicy to expect current serial algorithms to run faster in future. Even if the Processor cycles are drastically boomed, current compilers aren't simply clever enough yet to make it up to the multi core computation. In current technological scenario, a single fast core is being replaced by multiple slow ones.

- NP-completeness and approximation algorithms

- The alpha technique (e.g., disjoint sets, partial sums)

- FFT (Fast Fourier Transforms)

FunFact: Last yr's LEGO problem in MidTerms was FFT based.

1.4 Grading Policy

Four Homework Problem Sets: (The one with highest score is weighted 15%, lowest 5 % and others 10% each) {40% Net contribution}

- Per homework, two submissions are required. (Digital submission along with hardcopy handout)

- Homeworks must be typeset for them, to be checked against plagiarism. Implying, Scanned copies of handwritten content would be unacceptable.

- Submission can be in any format *(Word Doc, PDF)* but must be TYPESET accordingly.

Two exams: (The one with higher score would be weighed 30%, lower one 15%) {Net contribution towards final grade: 45%}

- MidTerm (in-class): Oct 11^{th}, 2017

- Final (in-class): Nov 29^{th}, 2017

(Open Notes, Open Slides, Homework Printouts are allowed. 75 mins, 75 Max pts to aim)

Scribe Notes: 10% (Everything discussed in class, doubt discussions, content on slides, Professor's jargon needs to get scribed. Submission of scribe notes must be made within a week post lecture).

Class participation & attendance: 5% (Participation includes but not limited to subtle laugh points with in-class humor and participation in doubt discussion. Everybody is recommended to attend the lecture as long as a seat is available).

Please Note: Classes don't terminate after final exam. There would be 2 more lecture sessions after Final Exam.

Special Note: Every source needs to be acknowledged in each submission towards the course. In essence, for a given problem even if you are able to find the crux of what exactly to look for from a source (except the exact solution), its pretty Cool!

1.5 Previous Year Grade distribution

Fall 2016 (158 students)

A: 17% A-: 30% B+: 37% B-: 6%

The distribution might not be the same everytime. It depends on the complexity of Problem sets and various other factors too.

1.6 Textbooks

Absolutely Required Textbooks

- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. Introduction to Algorithms (3rd Edition), MIT Press, 2009.

Recommended and somewhat required Textbooks

- Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms (1st Edition), McGraw-Hill, 2006.

- Jon Kleinberg and Eva Tardos. Algorithm Design (1st Edition), Addison Wesley, 2005.

- Rajeev Motwani and Prabhakar Raghavan. Randomized Algorithms (1st Edition), Cambridge University Press, 1995.

- Vijay Vazirani. Approximation Algorithms, Springer, 2010.

- Joseph JaJa. An Introduction to Parallel Algorithms (1st Edition), Addison Wesley, 1992.

1.7 What is an Algorithm

A well-defined computational procedure that solves a well-specified computational problem. It accepts a value or set of values as input, and produces a value or set of values as output in a **finite** amount of time.

Example: Mergesort solves the sorting problem specified as a relationship between the input and the output as follows.

Input: A sequence of *n* numbers $< a_1, a_2, ..., a_n >$

Output: A permutation $< a_1, a_2, ..., a_n >$ of the input sequence such that $a_1 \leq a_2 \leq ... \leq a_n$.

1.8 Desirable Properties of an Algorithm

- Correctness:

Designing an incorrect algorithm is straightforward. *An $\mathcal{O}(1)$ constant time algorithm to count number of students in a class is essentially of no use if it outputs the same result everytime.*

An incorrect algorithm could be useful at times when the error rate is controllable (e.g. Randomized algorithms).

-Efficiency:

Easily achievable if we give up on correctness.

Note that: $\exists$ a tradeoff between correctness and efficiency and certain classes of algorithms effectively exploit the middle ground between those.

- Randomized algorithms (Monte Carlo: always efficient but sometimes incorrect, Las Vegas: always correct but sometimes inefficient)

- Approximation algorithms (always incorrect!)

Exercise 1.8.1 *Given n numbers, find a number N larger than atleast $\frac{n}{2}$ of them. Thus, when sorted in non-decreasing order, the number N should exist in the upper half amongst the given set.*

Solution: Sorting would not just be an inefficient solution but also a redundant one since the problem does not demands a total ordering on the given set.

In the most optimal scenario, we would need to take a look at atleast $\frac{n}{2} + 1$ numbers.

But, we can further increase the efficiency if we giveup a bit on correctness. An $\mathcal{O}(1)$ algorithm which outputs any random number from the set would be 50% efficient. Lets take a look if we can increase the efficiency further.

Consider the scenario, when only 1 number is chosen. P(Success): $\frac{1}{2}$. P(failure): $\frac{1}{2}$

When 2 numbers are chosen,

P(Failure): $P(Failure, Num_1) * P(Failure, Num_2)$: $\frac{1}{4}$.

$\implies P(Success) : 1 - P(Failure) = \frac{3}{4}$.

Similarly, When k numbers have been chosen, P(Failure): $\frac{1}{2^k}$. $\implies$ P(Success): $1 - \frac{1}{2^k}$.

Note that : If we chose k such that $k = c.\log_2 n$

$\implies k = \log_2 n^c$

Thus, P(Success) = $1 - \frac{1}{2^k} = 1 - \frac{1}{2^{\log_2 n^c}}$

For large enough n, P(Success) approaches 1. Now, our algorithm is always efficient and sometimes incorrect.

*Such class of algorithms are known as **Monte-Carlo algorithms** which fall under the class of Randomized algorithms.*

But $\exists$ another class of **Approximation algorithms**, which are often used sometimes when we just don't have the demanding space, time or energy required for processing! *For instance*, a cellular GPS shouldn't take an hour to reflect the next turn on the road ahead and rather should be really fast. Thus, if the fastest route towards destination takes an hour in the most optimal case, an algorithm which comprises with a 5 min lapse to the destination and improves significantly on performance by giving up on efficiency would rather be preferred.

Thus, a small amount of tolerance in algorithms is often appreciated. They allow the program to carry computations even with limited available resources.

1.9 How to Measure Efficiency

Time is not always the only factor which we have a budget on. We are often limited on various other resources too which should be taken under consideration while designing an efficient algorithm. Some of these measures include:

- Space Complexity

- Cache Complexity

- I/O Complexity

- Energy Usage

- Number of processors/cores used

- Network bandwidth

- Time complexity

A note on importance of Cache complexity: Quite often, through the process of optimizing over performance a number of factors are overlooked. *For instance, the time taken for Data to reach the Processor from RAM.* Here, even if the Processors outrun the RAM with order of magnitudes of Giga-speeds, the cores would just end up sitting idle. To solve the problem, the systems are equipped with a quickly accessible small memory unit known as *Cache* for data that is accessed a lot frequently by the system. Now, what if we directly increase the size of cache memory to be comparable to that of a standard RAM and bypass the RAM all over to increase the net data access speed. Seems like a good idea but this won't simply work. If the cache size is increased to the order of Gigabytes as in a standard issue commercial RAM, it would take a lot more time for the byte signal to travel all the way through this cache towards the processor which would essentially make the cache speed comparable to a stock RAM.

1.10 Goal of Algorithm Analysis

The goal of analyzing an algorithm is to predict the behavior of that algorithm without implementing it on a real machine. But predicting the exact behavior is not always possible as there are too many influencing factors. Different programmers could have varied implementations of same algorithm, There could be different compilers putting together the code and mostly different processors behave differently with respect to their architecture. We thus model the machine first in order to analyze runtimes.

But an exact model will make the analysis too complicated! So we use an approximate model (e.g., assume unit-cost Random Access Machine model or RAM model). Furthermore, we may need to approximate even further: e.g., for a sorting algorithm we may count the comparison operations only. Various other factors could influence the runtime by constant amount too.

Crux: So the predicted running time will only be an approximation!

1.11 Performance bounds

Keeping all factors aside, we are generally interested in an algorithm's runtime. The runtime could be bound in a number of ways, the most important and common being the worst case analysis. Several other bounds

are listed as follows:

- Worst-case complexity: Maximum complexity over all inputs of a given size

- Average complexity: Average complexity over all inputs of a given size

- Amortized complexity: Worst-case bound on a sequence of operations

Thus, an amortized complexity x number of operations would be the worst case bound over that sequence of operations.

- Expected complexity: For algorithms that make random choices during execution (randomized algorithms)

To clear it further, we consider a case:

Let there be a box of n balls such that n_r number of balls are Red and n_b, blue. $(n = n_r + n_b)$

Using standard rules of Probability, we see that P(Red) = $\frac{n_r}{n_r+n_b}$*. Thus, if n = 1000,* $n_r = 10$*, then P(Red) =* $\frac{1}{100}$.

$\implies$ The expected number of times ones needs to try for a correct output = 100.

- High-probability bound: when the probability that the complexity holds is $\geq 1 - \frac{c}{n^\alpha}$ for input size n, positive constant c and some constant $\alpha \geq 1$

$\triangle$ End of Notes $\triangle$