Lecture 1 Note Monday, 01/31/2022

Announcements:

- **Room Change**: Our in-person class will be hold in King Hall 204 starting Wednesday. The building is very close to the main entrance.

- Please download and read the syllabus from Blackboard if you were not in class when we went over it.
- For students who didn't attend the class, make sure you attend at least one of the first five in person classes to avoid WN grade and being withdrawn by registrar.
- The Lecture PowerPoints will not be uploaded, instead, I will upload Lecture notes that include all the PowerPoint content and explanation of what we did in the class.
- The Lecture Recordings will be available on the following YouTube Playlists Link: https://youtube.com/playlist?list=PLZaTmV9UMKlgYpo2cAiMaEWxqyvbiXDFd

Go over the syllabus

Basics of Algorithm Analysis

References:
Algorithm Design* - Chapter 2 section 1,2,4
Introduction to Algorithms – Chapters 1, 2, and 3
Discrete Mathematics and its application – Chapter 3

Overview

- What is an algorithm?
  - [Intro to Algorithms] Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output.
  - A tool for solving a well-specified computational problem.

- What is analysis of algorithm?
  - The theoretical study of computer-program performance and resource usage.
    - When we talk about performance or resource usage, we normally refer to rum time and memory.
  - What's more important than performance?
    - correctness, simplicity, modularity, functionality, maintainability, reliability, extensibility, robustness, user-friendliness, programmer time, etc.

- Why study algorithms and performance?
  - Performance is currency of computing.
  - Algorithms help us to understand scalability.
  - Performance often draws the line between what is feasible and what is impossible.
  - Algorithmic mathematics provides a language for talking about program behavior.
  - The lessons of program performance generalize to other computing resources.
  - Speed is important. We always want to improve the speed, so we can solve the problem (algorithm) as fast as possible.

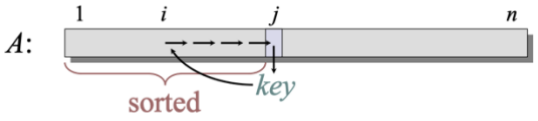
Example of Algorithm: Sorting

- It's a very simple problem, and one of the oldest problems that has been studies in algorithm. You want to put a sequence of number in monotonically increasing order.

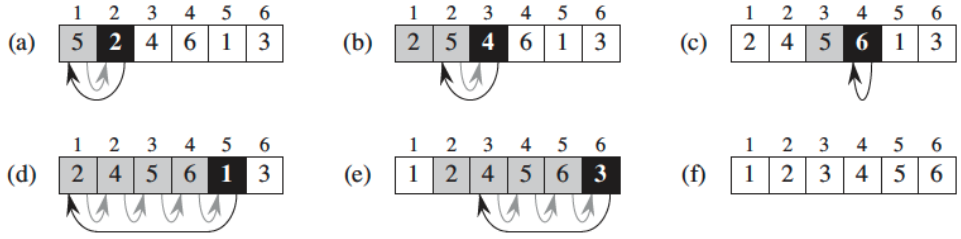
- Input: sequence $\langle a_1, a_2, a_3, \ldots, a_n \rangle$ of numbers.
- Output: permutation $\langle a'_1, a'_2, a'_3, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2, \leq a'_3 \leq \cdots \leq a'_n$.
  - There are many ways/algorithms to solve this problem, let's look at the insert sort.

- Insertion Sort

```
Insertion-sort(A, n):
   for j = 2 to n
      key = A[j]
      i = j-1
      while i > 0 and A[i] > key
         A[i+1] = A[i]
         i = i-1
      A[i+1] = key
```



- Example of Insertion Sort:
  Sort the array A = [5,2,4,6,1,3]



Analysis of Insertion Sort

- Running Time
  - Depends on input (e.g., already sorted)
  - Depends on input size (e.g., 6 elements or 6 million elements)
    - Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - General, we want to find upper bounds on the running time, because it represent a guarantee to the user.
    - Here is a program runs at most 3 seconds vs at least 3 sec.
    - Worst-case give you real information that you can used.

Kinds of Analysis

- Worst-case (usually)
  - $T(n)$ = maximum time of algorithm on any input of size n.
  - We usually want to find the worst case (the maximum run time), since it gives us a guarantee. If a problem has a runtime of at most 3 seconds, we know it will be done in 3 seconds.
- Average-case (sometimes)
  - $T(n)$ = expected time of algorithm over all inputs of size n.
  - Need assumption of statistical distribution of inputs.
  - This normally are used when your algorithm involves some random components to it, then we will look for the expected time of the algorithm.
- Best-case (barely)
  - Cheat with a slow algorithm that works fast on some input.
  - We generally want to avoid the best-case, since it might only work on some cases, and it doesn't tell you much about the algorithm. For example, an algorithm with at least 3 seconds running time, can take 3 hours, 3 days, even 3 years, there is no guarantee to it.

Analysis of Insertion Sort

- What is insertion sort's worst-case time?
  - It depends on the speed of our computer:
    - relative speed (on the same machine)
    - Absolute speed (on different machines)
    - Is one algorithm actually better no matter what machine it's run on? How can we talk about the worst-case time of an algorithm of a piece of software when I am not talking about the hardware?
      Clearly, if I had run on a faster machine, my algorithms are going to go faster.
- Big Idea: Asymptotic Analysis
  - Ignore machine-dependent constants.
  - Look at growth of $T(n)$ as $n \to \infty$

- Example:
  If it takes 25 low-level machine instructions to perform one operation in our high-level language, then our algorithm that took at most $1.62n^2 + 3.5n + 8$ steps can also be viewed as taking $40.5n^2 + 87.5n + 200$ steps when we analyze it at a level that is closer to the actual hardware.

Running Time

- The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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

Asymptotic Notations

- O-notation (Asymptotic Upper Bounds)
  - We say that $T(n)$ is $O(f(n))$ (read as "$T(n)$ is order $f(n)$") if, for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.
  - More precisely, $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

- Ω-notation (Asymptotic Lower Bounds)
  - $T(n)$ is $\Omega(f(n))$ if there exist constants $\epsilon > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \geq \epsilon \cdot f(n)$.

- Θ-notation (Asymptotically Tight Bounds)
  - If we can show that a running time $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, then in a natural sense we've found the "right" bound: $T(n)$ grows exactly like $f(n)$ to within a constant factor.
  - $T(n)$ is $\Theta(f(n))$ if there exist constants $c > 0$, $\epsilon > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $0 \leq \epsilon \cdot f(n) \leq T(n) \leq c \cdot f(n)$.
  - Normally, we can just drop the low-order terms and ignore the leading constants.

- o-notation
  - $T(n)$ is $o(f(n))$ if $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$.
  - If $T(n)$ is $o(f(n))$, then $T(n)$ is $O(f(n))$ but not $\Theta(f(n))$.

- ω-notation
  - $T(n)$ is $\omega(f(n))$ if and only if $f(n) = o(T(n))$.
  - In other word, if $\lim_{n\to\infty} \frac{f(n)}{T(n)} = 0$.
  - If $T(n)$ is $\omega(f(n))$, then $T(n)$ is $\Omega(f(n))$ but not $\Theta(f(n))$.

Review of Growth of functions

- Show that $f(n) = 3n^2 + 4n + 2$ is $O(n^2)$.

  We want to show that $3n^2 + 4n + 2 \leq c \cdot n^2$ whenever $n \geq n_0$ for some $c > 0$ and $n_0 \geq 0$.
  We know that $n \leq n^2$ and $1 \leq n^2$ for $n \geq 1$. Thus,
  $3n^2 + 4n + 2 \leq 3n^2 + 4n^2 + 2n^2 \leq 9n^2$ when $n \geq 1$.
  We have found that $c = 9$ and $n_0 = 1$ for $3n^2 + 4n + 2 \leq c \cdot n^2$.
  Therefore, $f(n) = 3n^2 + 4n + 2$ is $O(n^2)$.
  - The c and n_0 are not unique. You can find other c and n_0 that also satisfies the inequality. The small the c value, the bigger the n_0 will be. As long as you can find c and n_0 that works, you have shown that the function f is big-O of g.

- Show that $f(n) = 3n^2 + 4n + 2$ is $\Omega(n^2)$.

  For $\Omega(n^2)$, we want to show that $3n^2 + 4n + 2 \geq c \cdot n^2$ whenever $n \geq n_0$ for some $c > 0$ and $n_0 \geq 0$. And it's pretty clear that $3n^2 + 4n + 2 \geq 3n^2$ is always true.
  Then we can have $c = 3$ and $n_0 = 1$.
  You can also have $3n^2 + 4n + 2 \geq n^2$, then you will have $c = 1$ and $n_0 = 1$.
  Either way, you have shown that $f(n) = 3n^2 + 4n + 2$ is $\Omega(n^2)$.
  - Again, as long as you can find the c and n_0 that work, you have shown that the function f is big-Ω of g.

  Since $f(n) = 3n^2 + 4n + 2$ is both $O(n^2)$ and $\Omega(n^2)$, it's also $\Theta(n^2)$.

- Show that $f(n) = \frac{n^3 + 2n}{2n + 1}$ is $O(n^2)$.

  We want to show that $\frac{n^3 + 2n}{2n + 1} \leq c \cdot n^2$ whenever $n \geq n_0$ for some $c > 0$ and $n_0 \geq 0$. To make the functions bounded by $c \cdot n^2$, so we might want to get rid of the denominator. To do so, we want to make the numerator a multiple of the denominator and greater than $n^3 + 2n$ since we want to have upper bound. If we take $(2n + 1) \cdot n^2$, we will get $2n^3 + n^2$. $n^3 + 2n \leq 2n^3 + n^2$ for $n \geq 2$, since $n^3 \leq 2n^3$ and $2n \leq n^2$ when $n \geq 2$. Then we have $\frac{n^3 + 2n}{2n + 1} \leq \frac{2n^3 + n^2}{2n + 1} = \frac{(2n+1)\, n^2}{2n + 1} = n^2$ for $n \geq 2$. We found $c = 1$ and $n_0 = 2$.
  - This is slightly different with what we did in class, the goal is to find the c and n_0 that works.
  - When we work with fraction, to find upper bound of it (to find a something that's bigger than the fraction) you can either make the numerator larger or made the denominator smaller. To find the lower bound, you can do the vice versa.

  You can also do the following: $\frac{n^3 + 2n}{2n + 1} \leq \frac{n^3 + 2n}{2n} \leq \frac{n^2}{2} + 1 \leq \frac{n^2}{2} + \frac{n^2}{2} = n^2$ for $n \geq 2$. We also get $c = 1$ and $n_0 = 2$ here.

What to expect or prepare for the next class:

- We will continue working on different growth of functions. You can work the following questions on your own, we will go over them in next lecture.
- We will also review some basic logarithms and exponentiations operations.

- Show that $f(n) = \frac{n^3 + 2n}{2n + 1}$ is $\Omega(n^2)$.

- Prove that 2^n is not $O(n^2)$.

- Let k be positive integer. Show that $1^k + 2^k + \cdots + n^k$ is $O(n^{k+1})$.

- Let k be positive integer. Show that $1^k + 2^k + \cdots + n^k$ is $\Omega(n^{k+1})$.

- Arrange the following functions in increasing order:
  $(1.5)^n$, n^{100}, $(\log n)^3$, $\sqrt{n} \log n$, 10^n, $(n!)^2$, and $n^{99} + n^{98}$.

- Show that $\log(n!)$ is $\Theta(n \log(n))$.

- Solve for x and y in the following system of equations:
  $$\begin{cases} 2^{\frac{1}{3}\log_2 y^3} + 4^{\log_2 \sqrt{x}} = 5 \\ 8^{\log_4 (x^{2/3})} \cdot 9^{\log_9 y} = 6 \end{cases}$$

Reading Assignment

Algorithm Design: 2.1, 2.2, 2.4

Suggested Problems

- Algorithm Design – Chapter 2 – 1,2,3,4,5,8
- Discrete Mathematics and its Application
  - 3.1 - 9, 13, 19, 23, 27, 33, 41
  - 3.2 - 5, 9, 18, 21, 22, 24, 27, 33, 41, 42, 48