CS251 Midterm 1 - Fall 2022

schari

September 2022

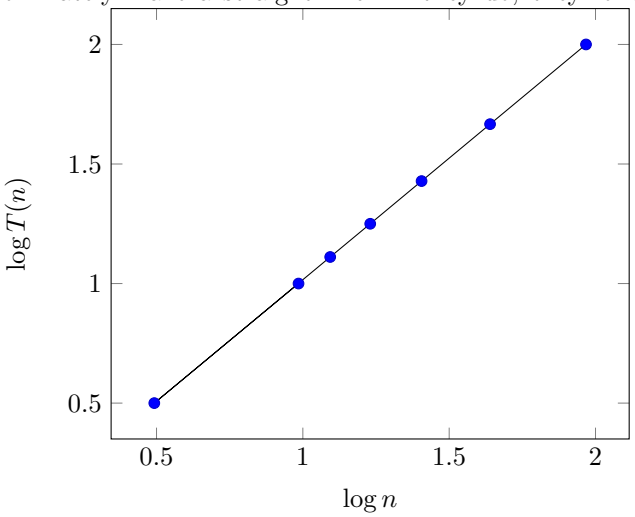
1 Summations and Logarithm Rules

- Summations
  - Given c is a constant, $\sum_{i=m}^{n} c = c(n - m + 1)$
  - $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$
  - $\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1)$
  - Given a function $f(i)$, $\sum_{i=m}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{m-1} f(i)$
- Log Rules
  - In CS 251, if you are just given a $\log(n)$ without a base, they probably mean $\log_2(n)$
  - $\log(ab) = \log(a) + \log(b)$
  - $\log(\frac{a}{b}) = \log(a) - \log(b)$
  - Given 2 numbers a and b, $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$
  - $\log(n^a) = a\log(n)$
  - $a^{\log_a(n)} = n$
  - $a^{b\log_a(n)} = n^b$

2 Experimental Analysis

- Limitations
  - Different machines can vary the run time
  - other processes/noise
  - May not be precise all the time
- These limitations are all put together to form constants c, which are seen in just about any runtime cost. In the power law, they are lumped together into a proportionality constant a.

- The power law is the law that $T(n)$ approximately follows the equation $T(n) = a * n^b$. The coefficient a will be based on the hardware, which you cannot control. However, the value of b is dependent on the algorithm.
- Given a table that contains data for n and the associated $T(n)$'s, we can check if the data follows the power law.
- Firstly, you can check if the values follow the power law by identifying the ratios between the rows of $T(n)$ and n. If the ratio between the rows of n are consistent, then the ratios between the rows of $T(n)$ should also be consistent (though it can be a different ratio between n and $T(n)$ and be okay. They just have to be consistent with their own rows).
  - If the ratios between rows of n are not consistent, that doesn't mean it doesn't follow the power law, only that its harder to check. Check the value of $\log_{r_n}(r_{T(n)})$ where r_n is the ratio between two rows of n and $r_{T(n)}$ is the ratio between rows of $T(n)$ (the same rows). For any two rows, this value should be essentially the same for it to follow the power law.
- Another way of identifying if the values follow the power law is to create a log-log plot. Plot the values of $\log n$ and $\log T(n)$ on a plot and determine if they approximately make a straight line. If they do, they follow the power law.



- Once it is identified that data follows the power law, a prediction can be made. If the next value of n given has the same ratio to a previous value, you can use the ratio of $T(n)$ to determine the predicted value of $T(n)$ very easily.
  - Given $T(8) = 16$ and $T(16) = 48$ and that $T(n)$ follows the power law, you can determine $T(64) = T(16*2^2) \approx T(16)*3^2 = 48*9 = 432$

(The 2 and 3 are determined from the ratios, $16/8 = 2$ and $48/16 = 3$.)

- You could be asked for the value of b in $T(n) = a * n^b$. You can determine this via
  1. Determining the slope of the log-log plot. That value is b.
  2. Take the logarithm of the ratio between rows of $T(n)$ with the base of the ratio of the rows of n. $b = \log_{r_n}(r_{T(n)})$.
     - For the given example of $T(8) = 16$ and $T(16) = 48$, $r_n = 16/8 = 2$ and $r_{T(n)} = 48/16 = 3$. Therefore, $b = \log_2 3$.

3 Recursive Functions

- Functions that call themselves in order to solve simpler problems
- Recursive functions don't call themselves infinitely; eventually stop when they reach a base case
-

4 Runtime Analysis

- Represents the efficiency of an algorithm
  - Usually represents the cost of an algorithm for a specific value of n.
  - However, can be amortized if the cost is distributed over an interval of values of n. (Equation is $\sum_{k=a}^{b} T(k)/(b-a+1)$)
- Three types of asymptotic runtime analysis: $O(n)$, $\Omega(n)$, and $\Theta(n)$
- $O(n)$
  - The asymptotic upper bound
  - Definition: Given functions $f(n)$ and $g(n)$, then $f(n) \in O(g(n))$ if there exists constants c and n_0 where $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
  - In other words, $f(n) \in O(g(n))$ if $f(n)$ doesn't grow faster than $g(n)$.
  - Growth order:
    - $O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < O(2^n) < O(n!)$
  - Going from the definition above, multiple functions can be big-O of another function.
    - Ex: $n \in O(n)$, and $n \in O(n^2)$
  - Generally, if they're asking for big-O of a function, you want to give the tightest bound of the function.

  - When giving the $O(n)$ of a function, you take the fastest-growing term and remove the constants from it
    * Ex: $4n^2 + 2n\log(n) + 3n + 123456 \in O(n^2)$

- $\Omega(n)$
  - Asymptotic lower bound
  - Definition: Given functions $f(n)$ and $g(n)$, then $f(n) \in \Omega(g(n))$ if there exists constants c and n_0 where $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$
  - In other words, $f(n) \in \Omega(g(n))$ if $f(n)$ doesn't grow slower than $g(n)$.
  - Like how multiple functions can be $O(n)$ of a function, multiple functions can also be $\Omega(n)$ of a function
    * Ex: $n \in \Omega(n)$, and $n \in \Omega(\log(n))$
  - Again, generally you should give the tightest bound
  - Process for getting big-Ω of a function is same as getting big-O

- $\Theta(n)$
  - Asymptotic tightest bound of a function
  - $f(n) \in \Theta(g(n))$ if $f(n)$ doesn't grow faster or slower than $g(n)$

- Asymptotic Growth Properties
  - If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then $f(n) \in \Omega(g(n))$ and vice versa
  - If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
  - If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$
  - If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$, then $f(n) \in \Theta(h(n))$
  - If $f(n) \in O(h(n))$ and $g(n) \in O(h(n))$, then $f(n) + g(n) \in O(h(n))$
  - If $f(n) \in \Theta(g(n))$, then $f(n) + g(n) \in \Theta(g(n))$
  - If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$

5 Algorithm Analysis

- Given an algorithm, we need to find the runtime cost of it.

- The runtime cost is based on what is specifically deemed worthy of noting. This can be the following:
  - The number of iterations.
    * for$(i = 0; i < n; i++)$//Do something
  - The value of a variable

    - $total = 0;$
      $\text{for}(i = 0; i < n; i++)\ total += i$
  - The number of times a function is ran.
    - $\text{for}(i = 0; i < n; i++) f(i)$ where $f(i)$ has a undefined cost correlating to the arguments.
  - The summed cost of function calls based on the arguments of the function.
    - $\text{for}(i = 0; i < n; i++) f(i)$ where $f(i)$ has a defined cost (e.g. $f(i) = i$ or $f(i) = \log i$).

- Some rules that you can follow are the following:
  - Loops and recursion generally turn into sigma notation.
  - If there is an "if" statement or other condition, it is usually better to seperate it into seperate parts or entire equations, instead of trying to keep it in the same parts. This includes things like the base case of a recursive function, if the base case counts as an iteration!
  - If the loop is going backwards, you can turn it around if it makes it easier. (e.g. you can turn subtraction into addition and swap upper and lower bounds, or division to multiplication), Just be careful its still the same iteration values!
  - If the comparison "$< f(n)$" is used, you probably want to use "ceil(f(n)) - 1" ($\lceil f(n) \rceil - 1$) as the upper bound. If its "$<= f(n)$", you probably want "floor(f(n))" ($\lfloor f(n) \rfloor$)
  - However, if looking at an asymptotic cost ($O(n)$, $\Omega(n)$, $\Theta(n)$), then you probably don't need floor and ceil at all, because they are miniscule compared to the entire sum.

- There are a number of ways of taking an algorithm and determining the cost.
  - Given a loop where a variable is incremented by 1 between each iteration, a simple sigma notation can be used.
    $\text{for}(i = 0; i < n; i++) f(i) \Rightarrow \sum_{i=0}^{n-1} T_f(i)$
  - Given a loop where a variable is multiplied by c between each iteration, you can use log, and then replace any value of i with 2^k.
    $\text{for}(i = 1; i < n; i* = c) f(i) \Rightarrow \sum_{k=0}^{\lceil \log_c n \rceil - 1} T_f(2^k)$
  - Given a recursive function that takes an input n, with base case a, and does a $f(n)$ cost operation, and then runs itself with value $n-1$, the summation would be: $\sum_{k=a}^{n} f(k)$
    Notice that this is exactly the same as a loop. For basic recursive functions, they are basically just loops. Anything else you might want induction.

- This is something we haven't covered, but it is useful in some circumstances. Given a loop where a variable is incremented by k between each iteration, you can stretch or squish the sigma accordingly.
  $\text{for}(i = a; i < n; i+ = k)f(i) \Rightarrow \sum_{i=0}^{\lceil (n-a)/k \rceil - 1} T_f(ki + a)$

6 Arrays and LinkedLists

- Arrays
  - A fixed size sequential data structure
  - Accessing data in an array $\in O(1)$
  - Inserting an element at the end of the array
    * If you keep track of the next available cell, $O(1)$
    * If not, $O(n)$
  - Inserting an element in the middle of a sorted array
    * Determining the position to insert the element + shifting elements to the right: $O(n)$
  - Resizing an array
    * Increasing the size of an array involves creating a new array with a larger size, then transferring the elements from the array to the new array
    * This process is $T(n) \approx n \in O(n)$
    * In order to minimize the amount of times we have to repeat the process of expanding the array, we generally double the size of the old array
    * Because this resizing is done so infrequently, we *amortize* this operation; that is, average it over several elements. Essentially, because we don't resize this array when inserting the first n elements, when the $n + 1^{\text{th}}$ element is inserted, despite resizing the array being relatively costly, we do it infrequently enough so that the average runtime cost is not as extreme.
    * Example proof: Let $T(k)$ be the runtime cost associated with resizing the array when inserting k elements into an array of size n. This means that the amortized cost $T(n) = \sum_{k=1}^{n+1} \frac{1}{n+1} T(k)$.

      When $k \leq n$, we do not resize the array, since the array still has capacity to accept new elements. Therefore, when $k \leq n$, $T(k) = 0$ since nothing is done to expand the array. Meanwhile, when $k = n + 1$, because we are resizing the array, we incur the full cost from resizing the array. Therefore, $T(n + 1) = n$.

Therefore,

$$\sum_{k=1}^{n+1} \frac{1}{n+1} T(k) = \frac{1}{n+1}(0 + 0 + \ldots + 0 + n) =$$

$$\frac{n}{n+1} = 1 - \frac{1}{n+1} \in O(1)$$

Therefore, the amortized runtime of $T(n) \in O(1)$.

* Here, I'm justifying why $T(k)$ evaluates the way it does at various values, then I'm taking the average of $T(k)$ at the various values I've specified.

- LinkedLists

7 Stacks

- Data structure to store and remove data
- Last data pushed into the stack would be the first data popped off (LIFO)
  - Think of it like a stack of plates; the last plate placed on top is the first plate taken from the stack
- Standard methods for stacks:
  - `push()` - Add an element to the top of the stack
  - `pop()` - Remove the element from the top of the stack
  - `isEmpty()` - Whether or not there are elements on the stock
  - `size()` - Number of elements on the stack
  - `peek()` - View the element at the top of the stack without removing it
- Implementation using Arrays vs LinkedLists
  - Arrays: Lower memory overhead; unable to resize to accomodate more elements
  - LinkedLists: Pointers require more memory; can expand to increase number of elements in the stack

8 Queues

- Data structure to store and remove data
- First data enqueued would be the first data dequeued (FIFO)

  - Think of it as a queue of people; first person to enter is also the first person who gets served

- Standard methods for queues:
  - `enqueue()` - Add an element to the end of the queue
  - `dequeue()` - Remove the element from the front of the queue
  - `isEmpty()` - Whether or not there are elements in the queue
  - `size()` - Number of elements in the queue
  - `peek()` - View the element at the front of the queue without removing it

- Implementation using Arrays vs LinkedLists
  - Arrays: In addition to lower memory overhead, a queue implementation will loop from the end of array to the front, and will keep track of the index of the front and back of the queue (see slides for example).
  - LinkedLists: Because LinkedLists are relatively resizable, you don't need to keep track of indices, but you will have to keep track of the front and back of the queue nodes.

9 Trees