Homework 1: Big-$\mathcal{O}$ and Introduction to Algorithms

Instructions: **Due 01/13/25 11:59pm**

- Please type your solutions using LaTex or any other software. Handwritten solutions will not be accepted.
- Please try to write concise responses.
- You should not use pseudocode to describe your algorithms.
- Unless otherwise stated, saying log means base 2

Q1 For the following list of functions, cluster the functions of the same order (f and g are of the same order if $f = \Theta(g)$), and then rank the groups in increasing order of magnitude. Recall $f = \Theta(g)$ if and only if $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$.

$f_1 = n\sqrt{n^5}$
$f_2 = n^{3.1415}$
$f_3 = 100n^{2^{\log(50)}}$ i.e. n to the power of (2 to the power of $\log(50)$)
$f_4 = 2^{2025}$
$f_5 = 5^{3\log_3(n)}$ i.e. 5 to the power of $3\log_3(n)$
$f_6 = 1024^{\log(n)}$
$f_7 = (\log(n))^{\log(n)}$
$f_8 = n^{\log(\log(n))}$
$f_9 = n\log(n) + 2025n!$

For example, an (incorrect) ordering could look like $f_1 < f_2 < (f_3 = f_4 = f_5) < f_6 < f_7 < (f_8 = f_9)$. Show your ordering, and justify any of the functions which you have put in the same order.

Firstly lets rewrite the functions as follows,

$f_1 = n^{5/2}n = n^{7/2} = n^{3.5}$
$f_2 = n^{3.1415}$
$f_3 = 100n^{2^{\log(50)}} = 100n^{50}$
$f_4 = 2^{2025}$
$f_5 = 5^{3\log_3(n)} = 125^{\log_3(n)}$ (as $3^4 < 125 < 3^5$ we have $n^4 < f_5 < n^5$)
$f_6 = 1024^{\log(n)} = (2^{10})^{\log(n)} = n^{10}$
$f_7 = (\log(n))^{\log(n)}$
$f_8 = n^{\log(\log(n))}$
$f_9 = n\log(n) + 2025n!$

Using this the ordering is as follows,

$$f_4 < f_2 < f_1 < f_5 < f_6 < f_3 < (f_7 = f_8) < f_9$$

To show $f_7 = f_8$. Consider,

$$\begin{aligned} &= \frac{f_7}{f_8} \\ &= \frac{(\log(n))^{\log(n)}}{n^{\log(\log(n))}} \\ &= \frac{2^{\log(n)\log(\log(n))}}{2^{\log(\log(n))\log(n)}} \\ &= 1 \end{aligned}$$

This means that $f_7 = f_8$ or that $f_7 = \Theta(f_8)$

Q2 True or False: An algorithm that only performs one `step` for each bit of N will take $\mathcal{O}(N)$ time to complete. Assume a `step` takes constant time. Carefully justify your answer in 1-3 sentences.

True. Assuming that N is a natural number we know that for any given N it can be represented as a binary with $\lceil \log(N) \rceil$ digits. So for instance 15 would require 4 digits. If our algorithm performs one step for each bit and we have $\log(N)$ bits then we will have $\log(N)$ operations. However we know that $\log(N)$ grows no faster than N so we have $\log(N) \in O(N)$ and our the answer is True.

Q3 Show that

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$$

Hint: To show an upper bound, decrease each denominator to the next power of 2. For a lower bound, increase each denominator to the next power of 2.

First we show upper bound.

We have,

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

Now we can upper bound this as follows by rounding each denominator to the next lowest power of two.

$$\sum_{i=1}^{n} \frac{1}{i} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \cdots$$

We can pair the terms in powers of two such that we can say that the total number of terms can be bounded as follows,

$$\begin{aligned} n &\leq 1 + \cdots + 2^{k-1} \\ &\leq 2^k - 1 \\ n + 1 &\leq 2^k \\ \log(n+1) &\leq k \end{aligned}$$

We can choose k as $k = \lceil \log(n) \rceil + 1$ as $\lceil \log(n+1) \rceil \leq \lceil \log(n) + 1 \rceil$ for larger values of n.

This also means that,

$$1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \cdots \leq 1 + \frac{1}{2} + \frac{1}{2} \cdots + \frac{1}{2^{k-1}} \leq k = \lceil \log(n) + 1 \rceil$$

So we have,

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i} &\leq k \\ &\leq \lceil \log(n) + 1 \rceil \\ &\leq 10 \lceil \log(n) \rceil \end{aligned}$$

So by taking $c = 10$ and for $n > 1$ we have,

$$\sum_{i=1}^{n} \frac{1}{i} = O(\log(n))$$

Now for lower bound we have,

$$\sum_{i=1}^{n} \frac{1}{i} \geq \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \dots$$

We group the terms similarly and we can bound them as follows,

$$2 + 2^2 + \cdots + 2^k \leq n$$

The sum of the geometric series gives us,

$$\begin{aligned} 2(2^k - 1) &\leq n \\ 2^k - 1 &\leq \frac{n}{2} \\ 2^k &\leq \frac{n}{2} + 1 \\ k &\leq \log(\frac{n+2}{2}) \end{aligned}$$

A possible value for k is $k = \lfloor \log(\frac{n}{2}) \rfloor = \lfloor \log(n) - 1 \rfloor$

So we get,

$$\begin{aligned} \lfloor \log(n) - 1 \rfloor &\leq \sum_{i=1}^{n} \frac{1}{i} \\ \lfloor \log(\frac{n}{2}) \rfloor &\leq \sum_{i=1}^{n} \frac{1}{i} \end{aligned}$$

Now take a value $c = \frac{1}{10}$ and we have,

$$\frac{1}{10} \lfloor \log(n) \rfloor \leq \lfloor \log(\frac{n}{2}) \rfloor \leq \sum_{i=1}^{n} \frac{1}{i}$$

The left can be shown as true by the following,

$$\begin{aligned} \log(n^{\frac{1}{10}}) &\leq \log(\frac{n}{2}) \\ n^{\frac{1}{10}} &\leq \frac{n}{2} \\ 2n^{\frac{1}{10}} &\leq n \\ 2^{10} &\leq n^9 \end{aligned}$$

We know this is true for any value of $n \geq 2$. Hence we have,

$$\lfloor \log(n) \rfloor \leq 10 \sum_{i=1}^{n} \frac{1}{i}$$

Where we take $c = 10$ and $n \geq 2$. Thus we have $\log(n) = O(\sum_{i=1}^{n} \frac{1}{i})$

Both of them give us,

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$$

Q4 Design an algorithm that fairly outputs a random integer between 1 and N. The only source of randomness you are allowed to use is a fair coin flip (heads or tails with 50% chance). Describe your algorithm in a few sentences, and justify why it works. You do not need to analyze the runtime of this algorithm.

Hint: store your integer in binary

First we store our integer in binary which means we have $\lceil \log(n) \rceil$ digits that are either 0 or 1 to represent our integer. Then we start from the left most digit and go toward the right most digit. We flip a coin and if the coin is heads we set it as 1 else we set it as 0. We go until the right most digit. Using the binary representation we are able to store number from 0 to $2^{\lceil \log(n) \rceil} - 1$. For perfect powers for instance this excludes the number. Hence we will add 1 to the number we get after this process. Now if the number generated after converting back to decimal is invalid, we restart the algorithm from beginning until we get a number smaller then equal to N and greater than equal to 1.

Because each bit has a $\frac{1}{2}$ chance of being either 1 or 0. Every number that can be represented using $\lceil \log(n) \rceil$ bits have an equal probability. This means that even if we disregard the numbers that are invalid the probability of the valid numbers still are mutually equal to each other. The idea being that if an invalid is generated and the algorithm is restarted, all the possible valid numbers will still have an equal probability of generation.

Q5 In class, we covered an algorithm to compute the nth Fibonacci number in $\mathcal{O}(2^n)$ steps. We're now going to attempt to do better than this. Notice that for some of the base cases of the Fibonacci recurrence, we can write it as a system of equations

$$\begin{aligned} F_1 &= 0 \cdot F_0 + 1 \cdot F_1 \\ F_2 &= 1 \cdot F_0 + 1 \cdot F_1 \end{aligned} \qquad \text{and similarly} \qquad \begin{aligned} F_2 &= 0 \cdot F_1 + 1 \cdot F_2 \\ F_3 &= 0 \cdot F_1 + 1 \cdot F_2 \end{aligned}$$

which are both equivalent to the matrix forms

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F_0 \\ F_1 \end{bmatrix} \qquad \text{and similarly} \qquad \begin{bmatrix} F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}.$$

We can substitute the first equation to the second to get

$$\begin{bmatrix} F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \begin{bmatrix} F_0 \\ F_1 \end{bmatrix} \qquad \text{which in general gives us} \quad \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}.$$

Notice how using this fact we can compute F_n. We simply take the nth power of the matrix, and multiply it with the vector $(F_0 \quad F_1)$. Thus, if we can take the nth power of the matrix quickly, we can compute the nth Fibonacci number quickly.

(a) Show that two 2×2 matrices can be multiplied using 4 additions and 8 multiplications.

Take any arbitrary 2×2 matrices as follows,

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

If we multiply the matrices out as AB we have,

$$\begin{bmatrix} a_1b_1 + a_2b_3 & a_1b_2 + a_2b_4 \\ a_3b_1 + a_4b_3 & a_3b_2 + a_4b_4 \end{bmatrix}$$

We see that for each entry of the four entires we require two multiplications and one addition. This means that to calculate all four entries we need $2 * 4 = 8$ multiplications and $1 * 4 = 4$ additions.

(b) Let X be any 2×2 matrix. Show that $O(\log(n))$ matrix multiplications suffice for computing X^n.

Hint: think about computing X^{16}

The general idea is that we do not need to do our matrix multiplication n times but because we can multiply matrices with itself. Our algorithm is as follows. If n is even then $X^n = X^{n/2} * X^{n/2}$ and $n = n/2$. If n is odd then $X^n = X^{n-1} * X$ and $n = n - 1$. The algorithm ends with $n = 1$. We see that the algorithms halfs the number of iterations required at least alternatively. This ensures that our algorithm performs the matrix multiplication using only $O(\log(n))$ matrix multiplications.

For instance take X^{16}. We are able to compute this by $M = X, M = X^2, M = X^4, M = X^{16}$. Which is 4 multiplications or $\log(16) = 4$ multiplications.

(c) Explain in 1-3 sentences why this fact doesn't necessarily mean that there will be an algorithm that can compute F_n in $O(\log(n))$ `steps`.

It is important to note that the number of multiplications and the number of steps required is different. For instance we see that in (a) that each multiplication requires 4 additions and 8 multiplications. Each of this counts as one operation. Now in addition to this when the numbers we have when N is very larger are large and require more bits to store that means that these operations of addition and multiplication will also be dependent on the number of bits themselves. So if we consider the operation of addition of multiplication on each bit as a singular step. So as n increases the number of steps on the bit level grows with n which can lead to higher complexities than $O(\log(n))$