CS 3510 Section C: Design & Analysis of Algorithms August 22nd 2023

HW 1: Big-$\mathcal{O}$ and Divide & Conquer

YOUR NAME HERE Due: August 29th 2023

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- Unless otherwise stated, all logarithms are to base two.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.

1.) (20 points) For the following list of functions, cluster the functions of the same order (i.e., f and g are in the same group if and only if $f = \Theta(g)$ into one group, and then rank the groups in decreasing order. You do not have to justify your answer.

(a.) $n\sqrt{n^5}$

(b.) $n^{3.1415}$

(c.) $100n^{2^{\log 50}} + n$ (Note: $\log 50$ is an exponent to 2)

(d.) 2^{2024}

(e.) $5^{3\log_3 n}$ (Note its 5 to the power of 3 $\log_3 n$)

(f.) $1024^{\log n}$

(g.) $(\log n)^{\log n}$

(h.) $n^{\log\log n}$

(i.) $n\log n + 2024n!$

(j.) $\log(n!)$

Solution: Increasing to Decreasing order:

1. $n^{\log\log n} = (2^{\log n})^{\log\log n} = (2^{\log\log n})^{\log n} = (\log n)^{\log n}$ so {h,g} is a group.
2. i is $O(n!)$
3. f is $1024^{\log n} = 2^{10\log n} = O(n^{10})$
4. e is $5^{\log_3 n^3} = (3^{\log_3 5})^{\log_3 n^3} = (3^{\log_3 n^3})^{\log_3 5} = n^{3\log_3 5} = O(n^{\log_3 125}) = O(n^{4.3949})$
5. d is $2^{2024} \implies O(1)$
6. c is $O(n^{50})$
7. b is $O(n^{3.1415})$
8. a is $O(n^{3.5})$
9. j is $O(n\log n)$

$\boxed{i > \{h, g\} > c > f > e > a > b > j > d}$

2.) (20 points) Suppose we had algorithms with the following run times. Answer the following questions and justify your answer.

(a.) $f(n) = n^{\mathcal{O}(1)}$
Could this function be exponential or bigger? Could it be polynomial?

> **Solution:** Since $\mathcal{O}(1)$ does not bound functions that depend on n this function cannot be exponential or larger than exponential. However, since $c = \mathcal{O}(1)$ for any constant c, this function could be polynomial.

(b.) $f(n) = n^{\omega(1)}$
Could this function be linear? Could it be polynomial? Could it be exponential or bigger?

> **Solution:** This function cannot be linear or polynomial, as $\omega(1)$ does not lower bound constants. However, it may be bigger than exponential, as $n = \omega(1)$.

(c.) $f(n) = 2^{\mathcal{O}(\log n)}$
Could this function be linear? Could it be polynomial? Could it be exponential or bigger?

> **Solution:** By log rules, $2^{\log_2(n)}n$. As a result, this function may be linear. Since $c \log n = \mathcal{O}(\log n) = \log n^c$, $2^{\log n^c} = n^c$, so this function may also be polynomial. However, this function cannot be exponential or bigger, since $\log n$ grows strictly less than n.

(d.) $f(n) = \mathcal{O}(2^{(\log n)^2})$
Could this function be linear? Could it be exponential or bigger?

> **Solution:** This function could be linear since $2^{(\log n)^2}$ upper bounds n. However, this function cannot be exponential as $(\log n)^2$ grows strictly less than n.

3.) (20 points) Assume n is a power of 4. Assume we are given an algorithm $f(n)$ as follows:

```
function f(n):
   if n>1:
      for i in range(15):
        f(n/4)
      for i in range(n*n):
        print("Banana")
      f(n/4)
   else:
      print("Monkey")
```

(a.) What is the running time for this function $f(n)$? Justify your answer. (Hint: Recurrences)

> **Solution:** $T(n) = 16T(n/4) + \mathcal{O}(n^2)$. $a = 16, b = 4, d = 2$, since $a = b^d$, we use case 2 of the Master's Theorem, The runtime is $\mathcal{O}(n^d \log n) = \boxed{\mathcal{O}(n^2 \log n)}$.

(b.) How many times will this function print "Monkey"? Please provide the exact number in terms of the input n. Justify your answer.

> **Solution:** The number of times the function prints "Monkey" is equal to the number of leaves in the recursive call tree. From drawing out the tree, there are a total of $\log_4(n)$ levels (since at each level we divide the subproblem size by 4. There is a factor of 16 new subproblems each level of the tree. Therefore, the total number of leaves is $\boxed{16^{\log_4(n)} = n^{\log_4(16)} = n^2}$.

4.) (20 points) Josh and Saigautam are trying to come up with Divide & Conquer approaches to a problem with input size n. Josh comes up with a solution that utilizes 6 subproblems, each of size $n/3$ with time n^2 to combine the subproblems. Meanwhile, Saigautam comes up with a solution that utilizes 8 subproblems, each of size $n/4$ with time $n\sqrt{n} + n$ to combine.

(a.) What is the runtime of both algorithms? Which one runs faster, if either?

> **Solution:** Josh's algorithm uses the recurrence $T(n) = 6T(n/3) + \mathcal{O}(n^2)$. Using the Master Theorem, we have $a = 6, b = 3, d = 2$. Since $a < b^d$, we use Case 1 of the Master Theorem which gives us $\mathcal{O}(n^d) = \mathcal{O}(n^2)$.
>
> Saigautam's algorithm uses the recurrence $T(n) = 8T(n/4) + \mathcal{O}(n^{3/2})$. Note that we disregard the term n as it is upper bounded by $n\sqrt{n}$. Using the Master Theorem, we have $a = 8, b = 4, d = 1.5$. Since $a = b^d$, we use Case 2 of the Master Theorem which gives us $\mathcal{O}(n^d \log n) = \mathcal{O}(n^{1.5} \log n)$.
>
> Since $\log n$ grows slower than $n^{0.5}$, Saigautam's algorithm is faster.

(b.) Let's say Diksha also tries to solve the same problem using an algorithm of her own. She utilizes 10 sub-problems of size $n/5$ with time $\log n$ to combine subproblems. Is this algorithm faster than the one you chose in part (a)? Why or why not?

> **Solution:** Yes. To avoid having to expand the recurrence, let's rewrite Diksha's algorithm's additional work as $\mathcal{O}(n)$. We have the recurrence $T(n) = 10T(n/5) + \mathcal{O}(n)$. We have $a = 10, b = 5, d = 1$. Since $a > b^d$, we have case 3 of the Master Theorem, which gives is $\mathcal{O}(n^{\log_5(10)})$ which is already smaller than $\mathcal{O}(n^{1.5})$. Since the modified algorithm with $\mathcal{O}(n)$ additional work is faster than Saigautam's algorithm, one that uses $\mathcal{O}(\log n)$ additional work will be even faster As a result, Diksha's algorithm is faster.

5.) (20 points) Assume that n is a power of two. The Hadamard matrix H_n is defined as follows:

$$H_1 = \begin{bmatrix} 1 \end{bmatrix}$$
$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$H_n = \begin{bmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{bmatrix}$$

Design an $O(n \log n)$ algorithm that calculates the vector $H_n v$, where n is a power of 2 and v is a vector of length n. Justify the runtime of your algorithm by providing a recurrence relation and solving it. (Hint: you may assume adding two vectors of order n takes $\mathcal{O}(n)$ time.)

Solution: Divide v into vertical halves v_T, v_B. Our product $H_n v$ will now look like this:

$$H_n v = \begin{bmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{bmatrix} \begin{bmatrix} v_T \\ v_B \end{bmatrix} = \begin{bmatrix} H_{n/2} v_T + H_{n/2} v_B \\ H_{n/2} v_T - H_{n/2} v_B \end{bmatrix}$$

The products $H_{n/2} v_T$ and $H_{n/2} v_B$ will output a vector of length $n/2$. These will be our two subproblems! We recursively calculate $H_{n/2} v_T$ and $H_{n/2} v_B$ for input H_n, v until we reach matrices of size 1, which will yield just a constant multiplication. To form the top half of the output vector $H_n v$, we add the result of $H_{n/2} v_T$ to $H_{n/2} v_B$, which is an addition of two vectors of length $n/2$. Similarly, to form the bottom half, we find the difference $H_{n/2} v_T - H_{n/2} v_B$. **Note that we don't have to compute 4 subproblems, as our subproblems are repeated in both rows. Once we find the two subproblems, we store them in variables and use them to find both the sum and difference**. The sum and difference take $\mathcal{O}(n)$ time, as we are adding two **vectors**, not numbers.

Our recurrence is now $T(n) = 2T(n/2) + \mathcal{O}(n)$, which is the merge sort recurrence. Using the Master Theorem, we have $a = 2, b = 2, d = 1$, which yields $\mathcal{O}(n \log n)$ time.