**CS 3510: Design & Analysis of Algorithms** August 25st 2024

# Problem Set 1: Big-$\mathcal{O}$ and Divide & Conquer

Wesley Lu Due: September 1th 2024

- 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. Pseudo-code 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 into one group (i.e., $f$ and $g$ are in the same group if and only if $f = O(g)$ and $g = O(f)$), and then rank the groups in increasing order of growth. You do not have to justify your answer.

(a.) $(\log n)^{\log n}$ :

*Logarithmic*

(b.) $n^{\pi}$

$$n^{3.14519}$$

*Polynomial*

(c.) $3^{2187}$

*Constant*

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

$$2^{\log_2(100)=100}$$

$$50n^{2^{\log_2(100)}} + n$$

$$50n^{100} + n$$

$$O(50n^{2^{\log 100}} + n) = O(n^{100})$$

This is because multiplicative constant are ignored and a polynomial dominates linear complexity.

*Polynomial*

(e.) $9^{3\log_3 n}$ (Note its 9 to the power of $3\log_3 n$)

$$9^{3\log_3(n)}$$

$$= 3^{2\cdot 3\log_3(n)}$$

$$= n^{2\cdot 3}$$

$$= n^{6}$$

*Polynomial*

(f.) $2048^{\log n}$

$$2048 = 2^{11}, 2^{11\log(n)}$$

$$= 2^{\log(n)\cdot 11}$$

$$= n^{11}$$

$$Polynomial$$

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

$$x = 2^{\log_2(x)}$$

$$n^{\log(\log n)} = 2^{\log(n)\log(\log n)}$$

$$= \log(n)^{\log(n)}$$

$$Logarithmic$$

(h.) $n\log(n^{15}) + 7n$

$$= n\log(n^{15}) + 7n$$

$$= 15n\log(n) + 7n$$

$$O(n\log(n^{15} + 7n) = O(n\log n)$$

This is because loglinear dominates linear complexity.

$$LogLinear$$

(i.) $n\sqrt{n^7}$

$$n^{\frac{9}{2}} = n^{4.5}$$

After simplifying using power rule, the equation becomes a polynomial.

$$Polynomial$$

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

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n$$

$$= \log(n!) = log(\sqrt{2\pi n}(\frac{n}{e})^n)$$

$$= \log(\sqrt{2\pi n}) + \log{(\frac{n}{e})^n}$$

$$= \frac{1}{2}\log(2\pi n) + n(\log(\frac{n}{e}))$$

$$= \frac{1}{2}\log(2\pi n) + n(\log(n) - \log(e))$$

$$= \frac{1}{2}\log(2\pi n) + n\log(n) - n(1)$$

Since $(n\log n)$ dominates $(n)$ and logarithm. $O(\log(n!) = O(n\log n)$

$$LogLinear$$

**Solution:** In increasing order, the time complexity for the following letters are: $[c < a\&g < h\&j < b < i < e < f < d]$

**2.)** (20 points) Suppose you are given the following inputs: a sorted array $A$, containing $n$ distinct integers, a lower bound $l$, and an upper bound $u$, where $l$ and $u$ might not be in the array $A$.

Design a divide & conquer algorithm in $\mathcal{O}(\log n)$ to find the number of elements in $A$ between $l$ and $u$, inclusive. Make sure to justify the algorithm's correctness and the time complexity (using Master's Theorem).

(a.) Describe your algorithm in plain english.

**Solution:** Perform binary search on the array A to find the upper bound $u$, and lower bound $l$. This will give the index L which represent the index for the value A[L] which is greater than or equal to the number $l$. Binary Search will also give the index U which represent the index for the value A[L] which is less than or equal to the number $u$. Given that L and U are inbound such that $L >= 0$ and $U < A.length$, then the number of element in range $[L, U]$ is equal to $U - L + 1$. This found in constant time by performing an addition operation.

(b.) Argue that your algorithm correctly transforms the input to the correct outputs.

**Solution:** The Binary Search algorithm will correctly find the lower and upper bound given that the array is sorted which is one of the conditions of the problem. The algorithm will handle edge cases where $l$ or $u$ is out of bounds. This will just set $l$ to zero or $u$ to the length of the array - 1. If there is no elements between the range, it will return 0.

(c.) Give the runtime of your algorithm and explain the reasoning.

**Solution:** The runtime of one binary search algorithm is $T(n) = T(n/2) + O(1)$ according to *Algorithms by Dasgupt, Papadimitriou, and Vazirani*. Using the master theorem, d=0, b=2, a=1 and $d = \log_b a, 0 = 0$. According to the Master Theorem, the time complexity of binary search is $O(n^d \log n) = O(\log n)$. Because we performed two binary search operations, the complexity would be $2 * log(n)$. However, in Big-O notation, it's standard to ignore multiplicative constant. Thus, the runtime of this algorithm will be in $O(\log n)$.

**3.)** (20 points) Assume $n$ is a power of 6. Assume we are given an algorithm $f(n)$ as follows:

```
function f(n):
   if n>1:
      for i in range(34):
        f(n/6)
      for i in range(n*n):
        print("Banana")
      f(n/6)
   else:
      print("Monkey")
```

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

**Solution:** The running time of this function is $O(n^2)$. Within this function, if $n > 1$, then it will recursively call $f(n/6)$ 34 times, loop $n * n$ times, and then call $f(n/6)$ 1 time. This result in time $T(n) = 34(\frac{n}{6}) + n^2 + \frac{n}{6}$. After simplifying, $T(n) = 35(\frac{n}{6}) + n^2$. Using the Master Theorem, $d > \log_b(a), 2 > 1.9843$ where $a = 35, b = 6, d = 2$. This means that $O(n^2)$ is the time complexity.

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

**Solution:** n is to the power of 6, where $n = 6^k$ for some integer $k$. For $n > 1$, the function will call f(n/6) 34 times. After, it will call f(n/6) one last time. This means that $M(n) = 34M(n/6) + M(n/6)$. The recursive function calls 35 times at each level of recursion. As $n$ is divided by 6 in each recursive call, the depth of the recursion tree is $\log_b n = \log_6 n$. At each leaf node of the tree, the where prints "Monkey". This means that it will be printed $n^{\log_6 35} = 35^{\log_6 n}$. (by logarithmic property)

**4.)** (20 points) Rohit and Saigautam are trying to come up with Divide & Conquer approaches to a problem with input size $n$. Rohit comes up with a solution that utilizes 27 subproblems, each of size $n/9$ with time $n\sqrt{n} + n\log n$ to combine the subproblems. Meanwhile, Saigautam comes up with a solution that utilizes 17 subproblems, each of size $n/4$ with time $n^2 + n\log^2 n$ to combine.

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

**Solution:**

Rohit:

$$T(n) = 27T(\frac{n}{9}) + n\sqrt{n} + n\log n$$

$$= 27T(\frac{n}{9}) + O(n^{3/2}) + \cancel{n\log n}$$

$$Master\ Theorem: d = \log_b(a),\ O(n^d \log n)$$

$$d = 3/2, a = 27, b = 9$$

$$\frac{3}{2} = \log_9 27,\ O(n^{3/2}\log n)$$

Saigautam:

$$T(n) = 17T(\frac{n}{4}) + n^2 + n\log^2 n$$

$$= 17T(\frac{n}{4}) + O(n^2) + \cancel{n\log^2 n}$$

$$Master\ Theorem: d < \log_b(a),\ O(n^{\log_b a})$$

$$d = 2, a = 17, b = 4$$

$$2 < \log_4 17 = 2.0437,\ O(n^{log_4 17}) = O(n^{2.0437})$$

Rohit will have a lower time complexity. Therefore, Rohit will have a faster running algorithm. $O(n^{3/2}\log(n)) < O(n^{2.04})$

(b.) Let's say Diksha also tries to solve the same problem using an algorithm of her own. She utilizes 11 sub-problems of size $n/6$ with time $\log n$ to combine subproblems. Is this algorithm faster than the one you chose in part (a)? Why or why not?

**Solution:**

Diksha:

$$T(n) = 11T(\frac{n}{6}) + O(\log n)$$

Since the master's theorem only works when there is a polynomial, I am going to assume $d = 0$.

$$d < \log_6(11) = \frac{\log 11}{\log_6} = 1.338$$

Using the Masters Theorem, the time complexity is $O(n^{1.338})$ Therefore, Diksha's algorithm is faster than (a) algorithm. In general, $O(n^{\alpha}) < O(n^{\alpha}\log n)$.

**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 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.)

(a.) Describe your algorithm in plain english.

**Solution:**

$$H_n(v) = \begin{bmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{bmatrix} \begin{bmatrix} v_{top} \\ v_{bottom} \end{bmatrix} = \begin{bmatrix} H_{n/2} \cdot v_{top} + H_{n/2} \cdot v_{bottom} \\ H_{n/2} \cdot v_{top} - H_{n/2} \cdot v_{bottom} \end{bmatrix}$$

This matrix is to calculate Hadmard Matrix multiplied by a vector $v$.
Base Case: The base case occurs where $n = 1$, $H_1 = 1$.
Recursive Case: Recursion happens when $n > 1$, the matrix $H_n(v)$ can be constructed using a $2x1$ matrix. The inputs are broken down into two equations as shown above. The algorithm solves two sub problems: $H_{n/2} \cdot v_{top} + H_{n/2} \cdot v_{bottom}$ and $H_{n/2} \cdot v_{top} - H_{n/2} \cdot v_{bottom}$. Combine the two into a single vector after.

(b.) Argue that your algorithm correctly transforms the input to the correct outputs.

**Solution:**

Base Case: For $n = 1$, the matrix is $[1]$ and the product of the product is just $H_1 \cdot v = v$. The base case is established for $n = 1$.
Inductive Hypothesis: Assume that the algorithm correctly computes for any k less than n where k is also a power of 2.
Inductive Step: Assume for n = 2k where k=n/2 the algorithm computes the two subproblems. By the inductive hypotheses will combine to form the vector $H_k v$. The result is perform according to the algorithm and matrix rule thus correctly computes for $H_n v$ for all powers of 2.

(c.) Give the runtime of your algorithm and explain the reasoning.

**Solution:** The algorithm recursively computes two functions $H_{n/2} \cdot v_{top} + H_{n/2} \cdot v_{bottom}$ and $H_{n/2} \cdot v_{top} - H_{n/2} \cdot v_{bottom}$. There is two sub-problems being computed of size n/2. Performing two vector additions of order $n$ takes $O(n)$ time. Therefore, the time is $T(n) = 2T(\frac{n}{2}) + O(n)$. Using Master Theorem, $a = 2, b = 2, d = 1$ and $\log_b a = \log_2 2 = 1$. Since $d = \log_b a$ where $1 = 1$, $T(n) = O(n \log(n))$.