CS 3510: Design & Analysis of Algorithms 01/12/2023

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

Professor Abrahim Ladha Due: 01/19/2023 11:59pm

- **Please TYPE your solutions using Latex or any other software. Handwritten solutions *won't* be accepted.**
- **Your solutions must be in plain English and mathematical expressions. Show the running time using the Master Theorem (wherever applicable).**
- **Unless otherwise stated, use *log* to the base 2.**
- **Unless otherwise stated, the fastest (and correct) algorithms will receive more credit. If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.**
- **Do not use pseudocode. Your answer will receive zero credit even if the pseudocode is correct.**

Name: Mohamed Ghanem

1.) (10 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 increasing order. You do not have to justify your answer.

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

(b.) $n^{1.0001}$

(c.) $n^3 + 3n^2 + n$

(d.) 2^{100}

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

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

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

(h.) $16^{\log \log n}$

(i.) $n \log n + 2023n!$

Solution:

For ease of ordering, begin by simplifying/manipulating the expressions given above:

(a.) $n\sqrt{n} \to n^{1.5}$

(b.) $n^{1.0001}$

(c.) $n^3 + 3n^2 + n \to n^3$

(d.) 2^{100} (constant)

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

(f.) $1024^{\log n} \to n^{10}$

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

(h.) $16^{\log \log n} \to \log^4 n$

(i.) $n \log n + 2023n! \to n!$

Increasing Order:

$$d < h < b < a < c < f < g < e < i$$

List Order:

1. 2^{100}
2. $16^{\log \log n}$
3. $n^{1.0001}$
4. $n\sqrt{n}$
5. $n^3 + 3n^2 + n$
6. $1024^{\log n}$
7. $(\log n)^{\log n}$
8. $n^{\log n}$
9. $n \log n + 2023n!$

2.) (15 points) Prove or disprove (with a counterexample) the following claims:

(a.) If $f(n) = \mathcal{O}(p(n))$ and $g(n) = \mathcal{O}(q(n))$, then
$f(n) - g(n) = \mathcal{O}(p(n) - q(n))$

Solution: Claim is **false**.

Let,

$$f(n) = n^2 + n \quad g(n) = n^2$$

Then,

$$p(n) = n^2 \quad q(n) = n^2$$

Note that,

$$p(n) = q(n) = n^2$$

Therefore,

$$\mathcal{O}(p(n) - q(n)) = \mathcal{O}(n^2 - n^2) = \mathcal{O}(1)$$

and,

$$f(n) - g(n) = n^2 + n - n^2 = n$$

Therefore the expression $f(n) - g(n) = \mathcal{O}(p(n) - q(n))$ evaluates to,

$$n = \mathcal{O}(1)$$

This statement is not true because the Big-O of n is not $\mathcal{O}(1)$. There exists no constant that satisfies the expression $n \leq c(0)$

Therefore for functions $f(n) = \mathcal{O}(p(n))$ and $g(n) = \mathcal{O}(q(n))$, $f(n) - g(n) = \mathcal{O}(p(n) - q(n))$ is not always true.

(b.) If $f(n) = \mathcal{O}(g(n))$, then $2^{f(n)} = \mathcal{O}(2^{g(n)})$

Solution: Claim is **false**.

Counter Example:
Let,

$$f(n) = 2n \quad g(n) = n$$

$$f(n) \leq cg(n)$$

$$2n \leq cn \text{ when } c \geq 2$$

Therefore,

$$f(n) = \mathcal{O}(g(n))$$

$f(n) = \mathcal{O}(g(n))$ can also be proven by showing that the ratio of between $f(n)$ and $g(n)$ is constant.

$$\frac{f(n)}{g(n)} = \frac{2n}{n} = 2$$

Therefore,

$$f(n) = \mathcal{O}(g(n))$$

Now check if $2^{f(n)} = \mathcal{O}(2^{g(n)})$ is true.

$$2^{f(n)} = 2^{2n}$$

$$2^{g(n)} = 2^{n}$$

$$2^{2n} \neq O(2^{n})$$

Once again, this can be proven by showing that the ratio of between $f(n)$ and $g(n)$ is not constant.

$$\frac{f(n)}{g(n)} = \frac{2^{2n}}{2^{n}} = 2^{n}$$

Therefore, $2^{f(n)} = \mathcal{O}(2^{g(n)})$ when $f(n) = \mathcal{O}(g(n))$, is not true for all $f(n)$ and $g(n)$

(c.) There are no functions $f(n)$ and $g(n)$ such that $f(n) \neq \mathcal{O}(g(n))$ and $g(n) \neq \mathcal{O}(f(n))$.

Solution: Claim is **false**.

Counter Example:
Not Monotonic and Not Piecewise:

$$f(n) = \sin(n) \quad g(n) = \cos(n)$$

Begin by proving $f(n) \neq \mathcal{O}(g(n))$.

Assume for the sake of contradiction that $sin(n) = \mathcal{O}(cos(n))$, that is, there exists a constant c and an n_0 such that for all $n > n_0$, $|sin(n)| \leq c|cos(n)|$.

Now, we can find a specific input n such that $|sin(n)| \leq c|cos(n)|$ does not hold, contradicting our assumption that $sin(n) = \mathcal{O}(cos(n))$ for all n.

One such input is $n = 2\pi k + \pi/2$ for any integer k. $cos(2\pi k + \pi/2) = 0$, and $\sin(2\pi k + \pi/2) = 1$. There exists no constant c that satisfies the inequality $1 \leq c * 0$.

Hence, our assumption that $sin(n) = \mathcal{O}(cos(n))$ for all $n > n_0$ is false, and we can conclude that $sin(n) \neq \mathcal{O}(cos(n))$, proving $f(n) \neq \mathcal{O}(g(n))$.

Next prove $g(n) \neq \mathcal{O}(f(n))$.

Assume for the sake of contradiction that $cos(n) = \mathcal{O}(sin(n))$, that is, there exists a constant c and an n_0 such that for all $n > n_0$, $|cos(n)| \leq c|sin(n)|$.

Now, we can find a specific input n such that $|cos(n)| \leq c|sin(n)|$ does not hold, contradicting our assumption that $sin(n) = \mathcal{O}(cos(n))$ for all n.

One such input is $n = 2\pi k$ for any integer k. $cos(2\pi k) = 1$, and $\sin(2\pi k) = 0$. There exists no constant c that satisfies the inequality $1 \leq c * 0$.

Hence, our assumption that $cos(n) = \mathcal{O}(sin(n))$ for all $n > n_0$ is false, and we can conclude that $cos(n) \neq \mathcal{O}(sin(n))$, proving $g(n) \neq \mathcal{O}(f(n))$.

This counter example works because of sine and cosine periodic nature. They intersect/overlap infinitely often and therefore can never be bounded by each other.

Therefore, there exists functions $f(n)$ and $g(n)$ such that $f(n) \neq \mathcal{O}(g(n))$ and $g(n) \neq \mathcal{O}(f(n))$.

3.) (10 points) Assume n is a power of 2. Assume we are given an algorithmic routine $f(n)$ as follows:

```
function f(n):
   if n>1:
      for i in range(n):
         for j in range(n*n):
            print("CS 1332")
      f(n/2)
      f(n/2)
   else:
      print("CS 3510")
```

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

Solution:

Master Theorem:

$$T(n) = aT([n/b]) + \mathcal{O}(n^d) \quad a > 0,\, b > 1,\, d \geq 0$$

$$f(n) = \begin{cases} \mathcal{O}(n^d), & \text{if } d > \log_b a \\ \mathcal{O}(n^d \log n), & \text{if } d = \log_b a \\ \mathcal{O}(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

In the algorithm:

(a) The branching factor (a) is 2 because each function call makes 2 recursive function calls.

(b) The size of the sub-problem (b) is 2 because at each function call the input n is divided by 2.

(c) The addition calculations (d) at each step is 3 because at every function call $O(n^3)$ work is done by the algorithm.

$$a = 2 \quad b = 2 \quad d = 3$$

The recurrence equation:

$$T(n) = 2T([n/2]) + \mathcal{O}(n^3)$$

Solving the recurrence equation using Master Theorem:

$$\log_b a = log_2 2 = 1$$

$$d > \log_b a \rightarrow 3 > 2$$

Runtime of algorithm:

$$\mathcal{O}(n^3)$$

(b.) How many times will this function print “CS 3510”? Please provide the exact number in terms of the input n. Justify your answer.

Solution:

The statement "CS 3510" is printed n times, assuming that the input n is a power of 2. This is because for any input n power 2 there are exactly $log_2(n)$ levels. The statement is printed whenever the base case is satisfied, which means the number of times the statement is printed is equal to the number of leaf nodes in the the recursive tree. The tree has a branching factor of 2, so the number of nodes at any level is 2^{level} This means that the number of nodes at the bottom most level (number of leaf nodes), is $2^{\log_2 n}$ which simplifies to n.

4.) (10 points) Suppose one of our TAs, Saahir, is working independently to solve a problem using Divide & Conquer. He designed a solution that breaks the problem into three sub-problems, each of size $n/3$, and combines the results in time $n^2 + n \log n + 1$. Another TA, Bharat, was able to solve the same problem using four sub-problems, each of size $n/2$, and combined the results in $\log n + 3n$ to get the final answer.

(a.) Write the recurrence relation and runtime of Saahir's algorithm.

Solution:

Master Theorem:

$$T(n) = aT([n/b]) + \mathcal{O}(n^d) \quad a > 0,\, b > 1,\, d \geq 0$$

$$f(n) = \begin{cases} \mathcal{O}(n^d), & \text{if } d > \log_b a \\ \mathcal{O}(n^d \log n), & \text{if } d = \log_b a \\ \mathcal{O}(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

In Saahir's algorithm:

(a) The branching factor (a) is 3 because the solution breaks the problem into three sub-problems.

(b) The size of the sub-problem (b) is 3 because each sub-problem is size $n/3$.

(c) The recombination complexity (d) is 2 because $\mathcal{O}(n^2 + n \log n + 1) = \mathcal{O}(n^2)$.

$$a = 3 \quad b = 3 \quad d = 2$$

The recurrence equation:

$$T(n) = 3T([n/3]) + \mathcal{O}(n^2)$$

Solving the recurrence equation using Master Theorem:

$$\log_b a = log_3 3 = 1$$

$$d > \log_b a \rightarrow 2 > 1$$

Runtime of Saahir's algorithm:

$$\mathcal{O}(n^2)$$

(b.) Write the recurrence relation and runtime of Bharat's algorithm.

Solution:

Master Theorem:

$$T(n) = aT([n/b]) + \mathcal{O}(n^d) \quad a > 0,\, b > 1,\, d \geq 0$$

$$f(n) = \begin{cases} \mathcal{O}(n^d), & \text{if } d > \log_b a \\ \mathcal{O}(n^d \log n), & \text{if } d = \log_b a \\ \mathcal{O}(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

In Bharat's algorithm:

(a) The branching factor (a) is 4 because the solution breaks the problem into four sub-problems.

(b) The size of the sub-problem (b) is 2 because each sub-problem is size $n/2$.

(c) The recombination complexity (d) is 1 because $\mathcal{O}(logn + 3n) = \mathcal{O}(n)$.

$$a = 4 \quad b = 2 \quad d = 1$$

The recurrence equation:

$$T(n) = 4T([n/2]) + \mathcal{O}(n)$$

Solving the recurrence equation using Master Theorem:

$$\log_b a = log_2 4 = 2$$

$$d < \log_b a \rightarrow 1 < 2$$

Runtime of Bharat's algorithm:

$$\mathcal{O}(n^{\log_b a}) = \mathcal{O}(n^{\log_2 4}) = O(n^2)$$

(c.) Determine whose algorithm was the fastest.

Solution:

Both algorithms have a runtime $\mathcal{O}(n^2)$, one algorithm is not faster than the other, they are both equally as fast.

5.) (15 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 provide the proof of correctness of your algorithm and the time complexity (using Master's Theorem).

Solution:

The goal of the algorithm is to find the indices of l and u in the sorted array A, then subtract the indices to find the number of values within the range of $[l, u]$. If l is not in the sorted array A, then the algorithm must find the index of the smallest element in the array that is greater than l. If u is not in the sorted array A then the algorithm must find the index of the largest element in the array that is less than u.

Note:
The Binary Search being returns the index of the target when the target is in the array. In the case where the target is not in the array, the search returns the "high" index variable instead of -1. This "high" index corresponds to the largest value in the array less than the target value. This constant time adjustment to the algorithm does not affect its O/ runtime complexity.

Proposed Algorithm:

1. If l is greater than u or the length of the sorted array A is 0 return 0.
2. Perform binary search on the array A for l, use the index i to search for the target l.
3. When the search is complete check the element at index i is equal to l, if not then the element at index i is the closest to l but less than it, so increment i to make sure that element is not included in the the final range calculation. Now the index i represents the smallest element in the array greater than the lower bound l.
4. Perform binary search on the array A for u, use the index j to search for the target u. Notice that if u is not in the array the range does not need to be adjusted, because the returned index j, already corresponds to the largest value in the array less than the target u.
5. Subtract index i from j and add 1 to obtain the number of values in the range $[l, u]$. The addition of 1 is to make sure that the result is inclusive.

Master Theorem for Binary Search:

1. The branching factor (a) is 1 because recursive call is made, at every binary search call.
2. The size of the sub-problem (b) is 2 because the at every stage in binary search half of the input is discarded.

3. There are no addition calculations for recombination (d) is 0 because there is constant additional work $\mathcal{O}(1)$.

$$a = 1 \quad b = 2 \quad d = 0$$

The recurrence equation:

$$T(n) = T([n/2]) + \mathcal{O}(1)$$

Solving the recurrence equation using Master Theorem:

$$\log_b a = log_2 1 = 0$$

$$d = \log_b a \rightarrow 0 = 0$$

Runtime of Binary Search algorithm:

$$\mathcal{O}(n^d \log n) = \mathcal{O}(\log n)$$

The algorithm consists of two binary searches, therefore its time complexity if the designed algorithm is 2 that of binary search.

$$2\mathcal{O}(\log n) = \mathcal{O}(\log(n))$$

Therefore the algorithm has a overall run time complexity of $\mathcal{O}(\log(n))$.

Proof of Correctness:

1. If u is less than l or the length of the sorted array A is 0, return 0, there is no valid solution.
2. If both l and u are in the sorted array A, the binary searches will find the indices of l and u, and the number of elements in the range is given by $j - i + 1$.
3. If l is not in the array but u is, the binary search for u will find its index, but the binary search for l will find the largest element less than l. In this case, the number of elements in the range is given by $j - (i + 1) + 1$.
4. If u is not in the array but l is, the binary search for l will find its index, but the binary search for u will find the largest element less than u. In this case, the number of elements in the range is given by $j - i + 1$ without any adjustments.
5. If both l and u are not in the array, the binary searches will not find the indices of l and u. In this case, the number of elements in the range is given by $j - i + 2$ after adding 1 to the result.

The algorithm handles all the possible input cases correctly, therefore the algorithm must be correct.