CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #1

Name(s): Raashi Maheshwari, Tanvi Yamarthy

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size *two* or *three* students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. This question reviews asymptotic notation. You may assume the following inequalities for this question (and throughout the course): For any constant $c \geq 1$,

$$(\log n)^c = o(n) \quad , \quad n^c = o(n^{c+1}) \quad , \quad c^n = o((c+1)^n) \quad , \quad (n/2)^{(n/2)} = o(n!).$$

(a) Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions $f_1, f_2, f_3, \ldots, f_9$ such that $f_1 = O(f_2)$, $f_2 = O(f_3), \ldots, f_8 = O(f_9)$. Remember to write down your proof for each equation $f_i = O(f_{i+1})$ in the sequence above. **(15 points)**

$\sqrt{n}$	$\log n$	$n^{\log n}$
$100n$	2^n	$n!$
9^n	$3^{3^{3^3}}$	$\dfrac{n}{\log^2 n}$

Hint: For some of the proofs, you can simply show that $f_i(n) \leq f_{i+1}(n)$ for all sufficiently large n which immediately implies $f_i = O(f_{i+1})$.

Solution.

$$\begin{aligned} f_1 &= 3^{3^{3^3}} \\ f_2 &= \log n \\ f_3 &= \sqrt{n} \\ f_4 &= \frac{n}{(\log^2 n)} \\ f_5 &= 100n \\ f_6 &= n^{\log n} \\ f_7 &= 2^n \\ f_8 &= 9^n \\ f_9 &= n! \end{aligned}$$

Proofs:

- $\lim_{n\to\infty} \frac{f_1}{f_2} = \lim_{n\to\infty} \frac{3^{3^3}}{\log n} = 0$
- $\lim_{n\to\infty} \frac{f_2}{f_3} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$ because $\log n = o(n)$
- $\lim_{n\to\infty} \frac{f_3}{f_4} = \lim_{n\to\infty} \frac{\sqrt{n}}{\log^2 n} = \lim_{n\to\infty} \frac{\log^2 n}{\sqrt{n}} = 0$
- $\lim_{n\to\infty} \frac{f_4}{f_5} = \lim_{n\to\infty} \frac{\log^2 n}{100n} = \frac{1}{100 \log^2 n} = 0$
- $\lim_{n\to\infty} \frac{f_5}{f_6} = \lim_{n\to\infty} \frac{100n}{n \log^2 n} = \frac{100}{\log^2 n} = 0$
- $\lim_{n\to\infty} \frac{f_6}{f_7} = \lim_{n\to\infty} \frac{n \log n}{2^n} = 0$
- $\lim_{n\to\infty} \frac{f_7}{f_8} = \lim_{n\to\infty} \frac{2^n}{g^n} = \lim_{n\to\infty} \left(\frac{2}{g}\right)^n = 0$
- $\lim_{n\to\infty} \frac{f_8}{f_9} = \lim_{n\to\infty} \frac{g^n}{n!} = 0$

(b) Consider the following four different functions $f(n)$:

$$1 \qquad \log\log n \qquad n^2 \qquad 2^{2^n}.$$

For each of these functions, determine which of the following statements is true and which one is false. Remember to write down your proof for each choice. **(10 points)**

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(\frac{n}{2}))$;
- $f(n) = \Theta(f(\sqrt{n}))$;

Example: For the function $f(n) = 2^{2^n}$, we have $f(n-1) = 2^{2^{n-1}}$. Since $2^{2^{n-1}} = 2^{\frac{1}{2}\cdot 2^n} = (2^{2^n})^{1/2}$.

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{n-1}}} = \lim_{n\to\infty} \frac{2^{2^n}}{(2^{2^n})^{1/2}} = \lim_{n\to\infty} (2^{2^n})^{1/2} = +\infty.$$

As such, $f(n) \neq O(f(n-1))$ and thus the first statement is false for 2^{2^n}.

Solution. 1. $f(n) = 1$

- $f(n) = \Theta(f(n-1))$: **TRUE**
  Proof: $f(n) = f(n-1) = 1$ for all n. Thus, $f(n) = \Theta(f(n-1))$.
- $f(n) = \Theta(f(n/2))$: **TRUE**
  Proof: $f(n) = f(n/2) = 1$ for all n. Thus, $f(n) = \Theta(f(n/2))$.
- $f(n) = \Theta(f(\sqrt{n}))$: **TRUE**
  Proof: $f(n) = f(\sqrt{n}) = 1$ for all n. Thus, $f(n) = \Theta(f(\sqrt{n}))$.

2. $f(n) = \log\log n$

- $f(n) = \Theta(f(n-1))$: **TRUE**
  Proof: $\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = 1$ since the -1 part does not matter for large n values, therefore n and $n-1$ become arbitrarily close.
- $f(n) = \Theta(f(n/2))$: **TRUE**
  $\lim_{n\to\infty} \frac{f(n)}{f(n/2)} = 1$ since the constant becomes negligible compared to n. Essentially, it becomes the same as dividing n/n.
- $f(n) = \Theta(f(\sqrt{n}))$: **FALSE**
  $\lim_{n\to\infty} \frac{f(n)}{f(\sqrt{n})} = \lim_{n\to\infty} \frac{\log\log n}{\log\log(\sqrt{n})}$

  $$= \lim_{n\to\infty} \frac{\log\log n}{\log((\log n)/2)} = \lim_{n\to\infty} \frac{\log\log n}{(\log\log n - \log 2)} = 1$$

  However, this limit approaching 1 doesn't mean they're asymptotically equal. The difference between $\log\log n$ and $\log\log(\sqrt{n})$ grows very largely as n increases.

3. $f(n) = n^2$

- $f(n) = \Theta(f(n-1))$: **TRUE**
  $\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{n^2}{(n-1)^2} = 1$
- $f(n) = \Theta(f(n/2))$: **FALSE**
  $\frac{f(n)}{f(n/2)} = \frac{n^2}{(n/2)^2} = 4$
  This ratio is constant, but not 1, so $f(n) \neq \Theta(f(n/2))$.
- $f(n) = \Theta(f(\sqrt{n}))$: **FALSE**
  $\frac{f(n)}{f(\sqrt{n})} = \frac{n^2}{(\sqrt{n})^2} = n$
  This ratio grows unbounded, so $f(n) \neq \Theta(f(\sqrt{n}))$.

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

- $f(n) = \Theta(f(n-1))$: **FALSE**
  As shown in the example, $\frac{f(n)}{f(n-1)}$ grows unbounded.

- $f(n) = \Theta(f(n/2))$: **FALSE**
  $\frac{f(n)}{f(n/2)} = \frac{2^{2^n}}{2^{2^{n/2}}} = 2^{2^n - 2^{n/2}}$
  This grows unbounded as n increases.
- $f(n) = \Theta(f(\sqrt{n}))$: **FALSE**
  $\frac{f(n)}{f(\sqrt{n})} = \frac{2^{2^n}}{2^{2^{\sqrt{n}}}} = 2^{2^n - 2^{\sqrt{n}}}$
  This grows even faster than the previous case.

(c) For each of the following functions, state whether $f(n) = O(g(n))$ or $f = \Omega(g(n))$, or if both are true, then write $f = \Theta(g(n))$. No proofs required for this problem. **(10 points)**

(a) $f(n) = n^2 - 7n$ and $g(n) = n^3 - 10n^2$
(b) $f(n) = (\sqrt{n})^3$ and $g(n) = n^2 - (\sqrt{n})^3$
(c) $f(n) = n^{log_3(4)}$ and $g(n) = n\log^3(n)$
(d) $f(n) = 2^{\log_2(n)}$ and $g(n) = n$
(e) $f(n) = log^5(n)$ and $g(n) = n/\log(n)$
(f) $f(n) = 4^n$ and $g(n) = 5^n$
(g) $f(n) = \log_4(n)$ and $g(n) = \log_5(n)$
(h) $f(n) = n^3$ and $g(n) = 2^n$
(i) $f(n) = \sqrt{n}$ and $g(n) = \log^3(n)$.
(j) $f(n) = n\log(n)$ and $g(n) = n^2$

Solution.

(a)	$f(n) = n^2 - 7n$	$g(n) = n^3 - 10n^2$	$f(n) = O(g(n))$
(b)	$f(n) = (\sqrt{n})^3$	$g(n) = n^2 - (\sqrt{n})^3$	$f(n) = O(g(n))$
(c)	$f(n) = n^{\log_3(4)}$	$g(n) = n\log_3(n)$	$f(n) = O(g(n))$
(d)	$f(n) = 2^{\log_2(n)}$	$g(n) = n$	$f(n) = \Theta(g(n))$
(e)	$f(n) = \log^5(n)$	$g(n) = \frac{n}{\log(n)}$	$f(n) = O(g(n))$
(f)	$f(n) = 4^n$	$g(n) = 5^n$	$f(n) = O(g(n))$
(g)	$f(n) = \log_4(n)$	$g(n) = \log_5(n)$	$f(n) = \Theta(g(n))$
(h)	$f(n) = n^3$	$g(n) = 2^n$	$f(n) = O(g(n))$
(i)	$f(n) = \sqrt{n}$	$g(n) = \log^3(n)$	$f(n) = \Omega(g(n))$
(j)	$f(n) = n\log(n)$	$g(n) = n^2$	$f(n) = O(g(n))$

Problem 2. Your goal in this problem is to analyze the *runtime* of the following (imaginary) recursive algorithms for some (even more imaginary) problem:

(A) Algorithm A divides an instance of size n into 5 sub problems of size $n/5$ each, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

Solution. Please refer to the end of document to see the recursion trees for all these algorithms. We start off by coming up with recurrence relation for the algorithm. There are 5 sub problems of n/5 as well as an $O(n)$ time to merge the sub problems together. Therefore the recurrence relation is T(n)=5T(n/5)+$O(n)$. The tree has log(5n) levels and each one performs at $O(n)$ time. Using this information, the tightest asymptotic upper bound is: $O(nlogn)$.

(B) Algorithm Algorithm B divides an instance of size n into 5 sub problems of size $n/2$ each, recursively solves each one, and then takes $O(n^2)$ time to combine the solutions and output the answer.

Solution. The recursive tree is at the end of this pdf. Please refer there. The recurrence relation is T(n)=5T(n/2)+ $O(n^2)$. This is because there are 5 problems of size n/2, and then $O(n^2)$ to combine the solutions. Using the recurrence relation tree we see that there are log base 2 n levels, and using this if we find the Asymptotic Upper Bound, then we get $\log_2 5, or \mathrm{O}(\mathrm{n}^{2.32})$.

(C) Algorithm C divides an instance of size n into 5 sub problems of size $n/2$ each, recursively solves each one, and then takes $O(n^{0.8})$ time to combine the solutions and output the answer.

Solution. Please refer to the end of this pdf for the tree diagram. The recurrence relation for this algorithm is T(n)=5T(n/2)+ $O(n^{0.8})$. Similar to the Algorithm from B, we see that there are log base 2(n) levels, and using this if we find the Asymptotic Upper Bound, then we get $\log_2 5, or, \mathrm{O}(\mathrm{n}^{2.32})$.

(D) D divides an instance of size n into 2 sub problems, one with size $n/4$ and one with size $n/5$, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

Solution. Please refer to the end of the pdf for the tree diagram. The recurrence relation for this is T(n)= T(n/4)+T(n/5), as you are dividing an instance of size n into two sub problems of size n/4 and n/5. Then we also account for the time it takes to combine the sub problems together, $O(n)$. We can see from the tree that there are log base 4 n levels, and the asymptotic upper bound is $O(nlogn)$

(E) Algorithm E divides an instance of size n in to 3 sub problems of size $n-1$ each, recursively solves each one, and then takes $O(1)$ time to combine the solutions and output the answer.

Solution. Please refer to the bottom of the PDF to find the recurrence tree relations. This essentially describes the slow sort algorithm, which we will get into in question 4. The algorithm has 3 recursive calls, all running on arrays of size $n-1$, with a constant case of $O(1)$ for sorting when the size is less than 100. Thus, the recurrence relation is:

$$T(n) = 3T(n-1) + O(1).$$

We see that each level of the recursion tree will have 3^n more nodes, and thus, the asymptotic upper bound is given by:

$$O(3^n).$$

For each algorithm, write a recurrence for its runtime and *use the recursion tree method* to solve this recurrence and find the *tightest asymptotic* upper bound on runtime of the algorithm. **(25 points)**

Problem 3. Each of the algorithms below takes as input a positive integer n and then performs some steps. For each algorithm, state the running time of the algorithm as a function of n, using Θ-notation.
(15 points)

a)
- For $i = 1$ to $n/3$
  - $j \leftarrow 10$
  - While $(j \leq i/20)$
    - Do placeholder stuff that takes $O(1)$ time
    - $j \leftarrow j + 10$

Solution. The runtime of this algorithm is $O(n^2)$. The reason why this is the runtime is because we can see that we have a for loop and a while loop nested inside of it. The outer for loop goes from i=1 all the way to $(j \leq n/3)$. As an example, let us say that our n=600, then the outer for loop will run 20 times. then, the inner while loop runs only for the iterations where j¡=i/20, meaning it will run only once because 10¡=200/20, is the only case that it is true. We can see that though the number of times the inner loop runs is very small compared to the number of times that the outer loop runs, in extremely large cases of n, the inner loop will run more times. Thus, mathematically, it's bound by $O(n^2)$.

---

b)
- While $(n \geq 1)$
  - $n \leftarrow n/3$
  - do placeholder stuff that takes $O(1)$ time.

Solution. The runtime of this algorithm is $O(logn)$. The reason is because the algorithm contains a statement n = n/3, which keeps on executing until $n < 1$. The iterations of the loop aren't solely till n, instead till the number of times the division is possible to be executed, keeping in mind the while loop check.

---

c)
- While $(n \geq 2)$
  - $n \leftarrow \sqrt{n}$
  - Do placeholder stuff that takes $O(1)$ time.

Solution. The run time is $O(loglogn)$. This is because every time we reassign n, we are taking the square root of it. Therefore, we run an logarithmic amount of iterations in regards to log n itself, making $O(loglogn)$.

---

d)
- For $i = 1$ to n
  - For $j = 1$ to i^2
    - Do placeholder stuff that takes $O(1)$ time

Solution. The run time of this algorithm is $O(n^3)$. This is because the outer loop runs n times, and then the inner loop runs $O(n^2)$ times for each iteration of i. Thus, n multiplied by (n^2), will ultimately result in a runtime of $O(n^3)$.

---

e)
- For $i = 1$ to n
  - For $j = 1$ to $\log(i)$
    - For $k = 1$ to n
      - Do placeholder stuff that takes $O(1)$ time

Solution. Solution to part (e) goes here. The runtime of this algorithm is $O(n^2 logn)$. This is because the outer loop runs n times, the inner loop runs log n times, and the innermost loop runs n times as well. Since all of these loops are nested, the runtime would end up being $O(n^2 logn)$.

Problem 4. In this problem, we consider a non-standard sorting algorithm called the *Slow Sort*. Given an array $A[1:n]$ of n integers, the algorithm is as follows:

• **Slow-Sort**$(A[1:n])$:

1. If $n < 100$, run merge sort (or selection sort or insertion sort) on A.
2. Otherwise, run **Slow-Sort**$(A[1:n-1])$, **Slow-Sort**$(A[2:n])$, and **Slow-Sort**$(A[1:n-1])$ again.

We now analyze this algorithm.

(a) Prove the correctness of **Slow-Sort**. **(15 points)**

Solution. We can use induction to prove the correctness of the Slow Start Algorithm. Every induction proof starts with a base case, and in this algorithm, it states that for n¡100, a different kind of sort (merge/insertion/selection) is used to sort the array, and we know that this indeed is true, meaning that the base case is supported (holds).

Let us move on to the inductive step. In the inductive step, we want to assume that $n \geq 100$. For our case, lets say n=100. There are three recursive calls that will be taken for this portion. The first call sorts the elements up to n-1 of the array. It sorts from 1 to 99 elements. The second call sorts everything but the first index(element 0 of the array), so from element 2 to the 100. The second sort call does not sort the first element, but we need to keep in mind that this is already done in the first sort call. Once again, there is a call Sort(Array[1:99]) again (first element to n - 1 elements), and this ensures that the sorted portion of the beginning array is properly merged with the sorted part of the end of the array.

After these steps, we know that the first and last n - 1 elements are sorted and that the largest element would be present at index Array[n].

At the end of the three recursive calls in Slow Sort, we can see that the entire array is properly sorted, thus proving the correctness of the array.

(b) Write a recurrence for **Slow-Sort** and use the recursion tree method to solve this recurrence and find the *tightest asymptotic* upper bound on the runtime of **Slow-Sort**. **(10 points)**

Solution. We can start writing our recurrence relation by thinking about the case where n is less than 100. In this case, the run time will be constant, because we essentially have a fixed n at the worst case (99), and we know that our array will only be of that size. Thus, performing any of the sorting algorithms on it (merge/insertion/selection) will result in a run time of $O(1)$.

Now, we need to look at the cases where n is greater than or equal to 100. In this case, we must remember that we have three recurrence calls, all in T(n-1) time. All 3 of these recursive calls are sorting n-1 elements which is why that is there corresponding run time. Combining all of these calls together along with the constant call leads to: $O(1) + T(n-1) + T(n-1) + T(n-1)$. Adding these up results in $O(1) + 3T(n-1)$.

We can break down this recurrence using the recurrence tree method. We know that the root of the tree will have a runtime cost of $O(1)$ because of the initial constant array size sort step. This node will have 3 children that sort arrays of size $n-1$. Since this is a recursive tree, we start with 1 node which

splits into 3 separate nodes, and each of those 3 nodes splits into 3 different sub problems as per the recursive calls.

Let us take an example. For simplicity, let's assume our threshold is $n > 5$. We have an array $[1, 9, 2, 7, 3]$. We make three recursive calls that split the array into:

$$\text{Slow sort } [1, 9, 2, 7], \quad [9, 2, 7, 3], \quad \text{and } [1, 9, 2, 7] \text{ again.}$$

Next, we need to explore each of these sub problems or array splits and perform the 3 recursive calls again. Through this process, we can see how the logic of the recursive tree works. We start with one node, which splits into 3 sub problems, and each of these 3 sub problems splits into 3 sub problems of their own.

Thus, we go from 1 node, to 3 nodes, to 9 nodes. At each level of the tree, the problem size decreases by 1 because we're sorting one more position in the array. So, if your array has 100 terms, you will eventually go from:

$$1 + 3 + 3^2 + 3^3 + \ldots + 3^{(n-1)} \text{ nodes.}$$

Following this logic for a geometric sequence, we can see that the total working time of this algorithm is $O(3^n)$. This is clearly seen by the recursive tree.

Problem 5. Karatsuba multiplication reduces the time complexity of multiplying two n-digit numbers to $O(n^{\log_2 3}) \approx O(n^{1.585})$, using the recurrence:

$$T(n) = 3T(n/2) + O(n)$$

The algorithm splits the numbers as follows:

$$x = x_1 \cdot 10^{n/2} + x_0, \quad y = y_1 \cdot 10^{n/2} + y_0$$

and computes three products: $x_0 \cdot y_0$, $x_1 \cdot y_1$, and $(x_0 + x_1) \cdot (y_0 + y_1)$.

Two values you are multiplying might be n/2 + 1 digits each, rather than n/2 digits, since the addition might have led to a carry. For instance compute $(x_1 + x_0)(y_1 + y_0)$ for $x = 53$ and $y = 52$. Notice the carry and the extra digit. So the recurrence will be

$$T(n) = 2T(n/2) + T(n/2 + 1) + O(n)$$

Prove that this issue won't affect our analysis and we still $O(n^{\log_2 3})$ complexity. **(20 points)**

Solution. The original Karatsuba multiplication recurrence is

$$T(n) = 3T(n/2) + O(n)$$

The

$$T(n) = 3T(n/2)$$

part of the recurrence represents the three recursive multiplications that Karatsuba does that take a size of n/2. The

$$T(n) = O(n)$$

represents the extra operations of the additions and shifts.

With the modified version when an extra digit is introduced, the recurrence is now changed to be:

$$T(n) = 2T(n/2) + T(n/2 + 1) + O(n)$$

In this case,

$$T(n) = T(n/2)$$

represents the multiplication of the two numbers $(x_0 * y_0)$ and $(x_1 * y_1)$. In this case, the two parts of the multiplication are approximately of size n/2. The extra digit that came from the multiplication of $(x_0 + x_1) * (y_0 + y_1)$ is accounted for by:

$$T(n) = T(n/2 + 1)$$

The last part of the recurrence is

$$T(n) = O(n)$$

This linear time handles the shifts, additions, and subtractions that take place.

The

$$T(n) = T(n/2 + 1)$$

can be approximated to be

$$T(n) = T(n/2)$$

This is because the +1 in the problem can simply be disregarded. Adding an extra constant digit to the recursive aspects of it would not asymptotically change the value of the growth rate significantly. Thus, the recurrence relation essentially simplifies to:

$$T(n) = 3T(n/2) + O(n)$$

which is the same thing as the original Karatsuba recurrence.

In order to solve both the original Karatsuba recurrence and the modified Karatsuba recurrence (with the extra digit carry), we can use the master theorem. With the master theorem, we first calculate $\log_b(a)$ which is $\log_2(3)$ which approximates to 1.585. Then, we have to compare $\log_b(a)$ with d. In this case, d = 1 and $1.585 > 1$. Thus, using the master theorem, we can conclude that adding an extra digit would not affect the overall asymptotic complexity and we would still end up with: $O(n^{\log_2 3})$ complexity.

Problem 6. Toom-Cook multiplication generalizes Karatsuba by dividing numbers into more parts. The Toom-3 algorithm splits two n-digit numbers x and y into three smaller parts:

$$x = x_2 \cdot 10^{2k} + x_1 \cdot 10^k + x_0, \quad y = y_2 \cdot 10^{2k} + y_1 \cdot 10^k + y_0$$

If we multiply we need to perform 9 multiplications $x_i y_j$ of size $n/3$.

1. Write down the recurrence relation for the time complexity of the simple divide and conquer algorithm and solve it.
2. Toom-3 algorithm instead only uses 5 smaller multiplications to calculate all the 9 products needed. Find the time complexity of Toom-3.

(20 points)

Solution. (A) The recurrence relation for the time complexity of the simple divide and conquer algorithm would be:

$$T(n) = 9T(n/3) + O(n)$$

In this recurrence relation,

$$9T(n/3)$$

accounts for the nine recursive multiplications, each of size n/3. The

$$O(n)$$

accounts for the cost of additions and shifts.

To solve this recurrence relation, you can use the Master theorem since the recurrence relation has the form

$$T(n) = aT(n/b) + O(n^d)$$

In this case, the number of sub-problems is 9, hence a = 9 and the size of each sub-problem is 3, hence, b = 3. The value of the exponent that represents the merging part of the algorithm is 1, hence d = 1. When comparing the values of a, b and d, we can see that

$$a > (b^d)$$

Thus, using the master theorem, we can claim that the running time is

$$T(n) = O(n^{log_b(a)})$$

When we calculate that, we end up getting

$$O(n^{log_3(9)})$$

which equals 2. We can finally use that to conclude that the time complexity of the simple divide and conquer algorithm would be $O(n^2)$

(B) In order to find the time-complexity of Toom-3, we would use similar logic as the previous part. When using the Toom-3 algorithm, you perform 5 multiplications of size n/3. The cost of merging stays the same as

$$O(n)$$

This leads to a recurrence relation of:

$$T(n) = 5T(n/3) + O(n)$$

As the previous part, we can use the master theorem to solve this. In this case, the number of sub-problems is 5, the size of each sub-problem stays the same as 3 and the cost to merge stays as 1. Using those facts, we end up with the following values for the variables: a = 5, b = 3, and d = 1. When comparing the values of a, b and d, we can see that

$$a > (b^d)$$

Thus, using the master theorem, we can claim that the running time is

$$T(n) = O(n^{log_b(a)})$$

When we calculate that, we end up getting

$$O(n^{log_3(5)})$$

which is approximately 1.465. We can finally use that to conclude that the time complexity of the Toom-3 algorithm is: $O(n^{1.465})$ This algorithm is better than the simple divide and conquer algorithm as well as the Karatsuba's algorithm.

Problem 7. Merge Sort usually splits an array into two equal halves. However, consider a version where the array is split into parts of size $2n/3$ and $n/3$.

a) Write the recurrence relation for the time complexity of Merge Sort with this unequal split.

b) Solve the recurrence and find the time complexity.

(20 points)

Solution. (A) The recurrence relation for the time complexity of Merge Sort with this unequal split would be:

$$T(n) = T(2n/3) + T(n/3) + cn$$

which simplifies to

$$T(n) = T(2n/3) + T(n/3) + O(n)$$

The

$$T(2n/3)$$

represents the complexity for the larger part of the split,

$$T(n/3)$$

represents the time complexity for the smaller part, and the

$$O(n)$$

represents the time it takes to combine/merge the two arrays.

(B) Since the recurrence theorem is of the form:

$$T(n) = aT(n/b) + O(n^d)$$

we can use the master theorem or the recursion tree method to solve this problem. If we use the recursion tree method, we see that:
Level 0: cn

Level 1: $c(2n/3) + c(n/3) = cn$

Level 2: $c(4n/9) + c(2n/9) + c(n/9) = cn$

The pattern that is visible here is that at each level, the total work being completed is c*n. In order to find the time complexity, we would have to multiply that by the number of levels. The second element of the recurrence relation

$$(n/3)$$

tells us that the size reduces by a factor of 3 for each level. Thus, we can solve for k, the number of levels by doing the following:

$$n\left(\frac{1}{3}\right)^k = 1$$

$$\left(\frac{1}{3}\right)^k = \frac{1}{n}$$

$$k \log\left(\frac{1}{3}\right) = \log\left(\frac{1}{n}\right)$$

$$k = \log_3 n$$

We can see that the recursive tree has

$$\log_3 n$$

levels and we previously concluded that each level does c*n work. Thus, the total work/time complexity of the merge sort with the unequal split can be calculated as:

$$T(n) = cn \log_3(n) = O(n \log n)$$

The time complexity can be simplified to

$$O(nlogn)$$

This shows us that the unequal split does not change the time complexity because standard merge sort also has a time complexity of

$$O(nlogn)$$

Problem 8. Consider the problem of computing the Fibonacci numbers:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

a) Write an algorithm to compute the Fibonacci number F_n with $O(n)$ additions.

Note that F_n grows exponentially and has $\Theta(n)$ digits which means that the additions operation $F_n + F_{n+1}$ actually takes $O(n)$ time. Compute the time complexity of the algorithm.

b) We start by writing the equations $F_1 = F_1$ and $F_2 = F_0 + F_1$ in matrix notation:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

In general,

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

So, in order to compute F_n, it suffices to raise this 2×2 matrix, call it X, to the nth power.

i) Show that $O(\log n)$ matrix multiplications suffice for computing X^n by repeated squaring. Note that two 2×2 matrices can be multiplied using 4 additions and 8 multiplications.

ii) Let $M(n)$ be the running time of an algorithm for multiplying n-bit numbers. Prove that the running time of this second algorithm is $O(M(n) \log n)$.

(20 points)

Solution. (a) The pseudocode for the algorithm that can be used to compute the Fibonnaci number F_n with $O(n)$ additions is as follows:

1. Set $F_0 = 0$ and $F_1 = 1$.
2. For $i = 2$ to n:
   (a) Compute $F_i = F_{i-1} + F_{i-2}$.

3. Return F_n.

(b) Part i: The exponentiation by squaring technique is used to compute X^n more efficiently. The process starts by expressing the exponent n in its binary form. The method then involves repeated squaring of X (i.e., $X \times X$). If n is odd at any point, an extra factor of X is multiplied into the result since the current bit is 1. However, if n is even, no additional multiplication is required. This selective multiplication when n is odd helps reduce the total number of operations from n to about $\log n$, improving the runtime from $O(n)$ to $O(\log n)$.

To compute A^n efficiently using this method, follow these steps:

1. Set *result* to the identity element I and assign X to A.
2. While $n > 0$:
   (a) If n is odd, update *result* by multiplying it by X.
   (b) Square X by setting $X = X \times X$.
   (c) Divide n by 2 (ignoring the remainder. The floor function can be used for this).
3. Finally, return *result*.

(b) Part ii:

Let $M(n)$ denote the time required to multiply n-bit numbers. The total running time is $O(M(n) \log n)$.

This results from performing $O(\log n)$ matrix multiplications using the exponentiation by squaring method described earlier.

When multiplying two 2×2 matrices, typically 8 multiplications and 4 additions are needed. However, since we are working with bits (0 or 1), this can be optimized to just 4 multiplications and 2 additions.

As a result, each matrix multiplication requires $O(M(n))$ time, since the number of operations remains constant.

In summary, matrix multiplication on n-bit numbers can be completed in constant time, meaning it does not substantially affect the overall time complexity. The efficiency of large number multiplication becomes the key determinant in the algorithm's overall performance.

Challenge Yourself. We have an n-story building and a series of magical vases that work as follows: there is some unknown level L in the building that if we throw these vases down from any of the levels $L, L+1, \ldots, n$, they will definitely break; however, no matter how many times we throw the vases down from any level below L nothing will happen them. Our goal in this question is to determine this level L by throwing the vases from different levels of the building (!).

For each of the scenarios below, design an algorithm that uses asymptotically the smallest number of times we throw a vase (so the measure of efficiency for us is the number of vase throws).

(a) When we have only one vase. Once we break the vase, there is nothing else we can do. **(+2 points)**

Solution. When you only have one vase, you have to proceed with caution which means that you'll have to go level by level, one at a time. This ensures that you do not miss the unknown floor L from where the vase will break. You have to go one floor at a time because if the vase breaks once, you do not have any other vases to continue testing. Thus, the safest way to find the floor from which the vase will break is to go floor by floor. The algorithm for determining the floor from which the vase will break would be as follows:

(a) Start at level 1.

(b) Throw the vase from each subsequent level until the vase breaks.

(c) Once the vase breaks, the last level from which it did not break ($L-1$ is the unknown level L.)

Time Complexity:
In the worst-case scenario, you will throw the vase from every level, so the number of vase throws will be $O(n)$, where n is the number of levels in the building.

(b) When we have four vases. Once we break all four vases, there is nothing else we can do. **(+4 points)**

Solution. When you have four vases, you have more chances to continue testing. In this case, even if you break one vase, you still have three other vases to test with. Thus, instead of testing floor by floor, you can spread out the testing more consistently across the floors.

(a) You can divide the building into chunks and you can start by throwing the first vase from the k-th floor. If the vase breaks from that floor, we know that the critical level of L would be below the k-th floor. If it does not break, you can continue testing the floors above the k-th floor.

(b) When you throw the second vase, you separate the building into another chunk that is $k-1$ floors up. If the vase did not break, you can continue testing the higher floors as mentioned before. This process can repeat.

(c) When the vase breaks at some floor, use the remaining vases to search the previous chunk of floors below the level where the vase broke.

(d) The goal is to minimize the number of throws by progressively reducing the number of floors you need to check after each vase breaks.

The first throw should be from floor k, the second throw from $k+(k-1)$, the third from $k+(k-1)+(k-2)$, and so on. This ensures that after each vase break, you only need to check the remaining floors with the remaining vases.

To find the best value of k, the sum of floors you need to check should cover all n floors. This is described by:

$$k+(k-1)+(k-2)+\cdots+1 \geq n$$

which simplifies to:

$$\frac{k(k+1)}{2} \geq n$$

(a) Let k be the largest integer such that $\frac{k(k+1)}{2} \geq n$.

(b) Start from the k-th floor and throw the first vase.

(c) If it breaks, test every floor below this one with the remaining vases.

(d) If it doesn't break, move to the next group of floors, adjusting the size of each group as you progress.

(e) Repeat this process, reducing the group size after each vase breaks.

This approach returns a time complexity of $O(\sqrt{n})$, which is much better than the $O(n)$ time for when we had one vase.

(c) When we have an unlimited number of vases. **(+4 points)**

Solution. When we do not have a restriction on the number of vases, we an implement an algorithm similar to binary search for the n floors to find the minimum number of throws needed to find the level L

(a) Start with the middle floor of the building (floor $\lceil n/2 \rceil$).

(b) Throw a vase from this floor:

- If the vase breaks, restrict the search to the lower half of the floors because we know that level L has to be a lower floor.
- If the vase does not break, restrict the search to the upper half of the floors, as we know the level L is going to be a higher floor.

(c) This process repeats by continually throwing a vase from the middle of each "chunk"/"part" of the remaining floors. The parts/chunks of the floors keep getting cut in half after each throw.

(d) This implementation of the binary search algorithm will continue until the part/chunk of the floor is reduced to one singular floor, which is the level L.

Since the search space is halved with each throw (as this is an implementation of binary search), the number of times we throw the vase is $\log n$, where n is the number of floors in the building. Therefore, the time complexity of this algorithm is $O(\log n)$.

Figures attached below:

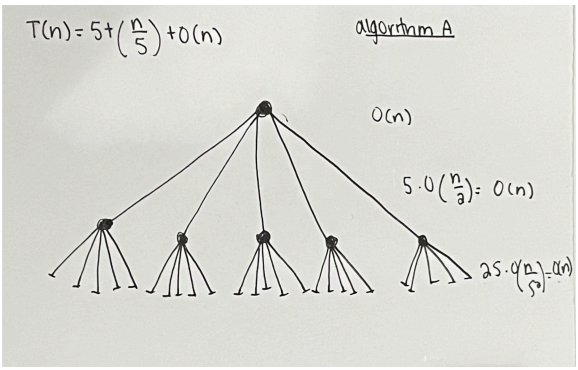

Figure 1: Recursion tree for algorithm A.

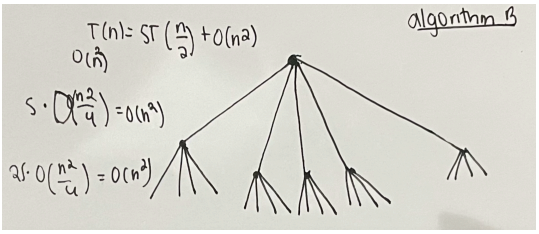

Figure 2: Recursion tree for algorithm B.

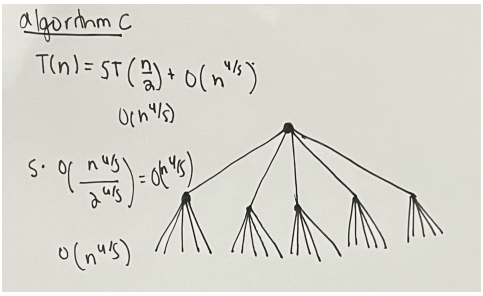

Figure 3: Recursion tree for algorithm C.

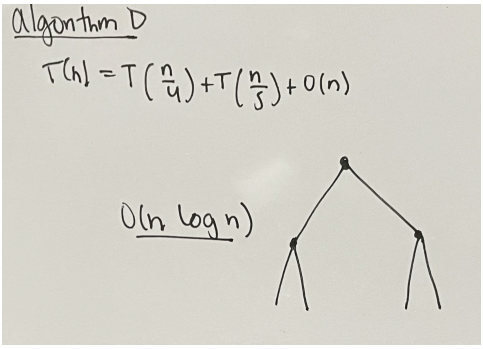

Figure 4: Recursion tree for algorithm D.

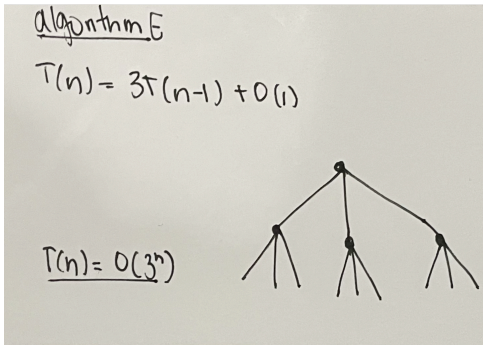

Figure 5: Recursion tree for algorithm E.