1

$$\begin{aligned}
\sum_{i=1}^{n} i(i+1) &= \sum_{i=1}^{n} (i^2 + i) \\
&= \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \\
&= \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \\
&= \frac{n(n+1)}{2} \left(\frac{2n+1}{3} + 1 \right) \\
&= \frac{n(n+1)}{2} \left(\frac{2n+4}{3} \right) \\
&= \frac{n(n+1)(2n+4)}{6}
\end{aligned}$$

2

We can use mathematical induction.

For the base case:

For $n = 1$:

- The Lucas number $L_1 = 1$.
- The Fibonacci numbers $F_0 = 0$ and $F_2 = 1$.
- So, $F_{1-1} + F_{1+1} = F_0 + F_2 = 0 + 1 = 1$, which equals L_1.

For $n = 2$:

- The Lucas number $L_2 = L_1 + L_0 = 1 + 2 = 3$.
- The Fibonacci numbers $F_1 = 1$ and $F_3 = 2$.
- So, $F_{2-1} + F_{2+1} = F_1 + F_3 = 1 + 2 = 3$, which equals L_2.

For inductive step:

Assume that the statement is true for some arbitrary positive integer k, such that $L_k = F_{k-1} + F_{k+1}$.

We need to show that $L_{k+1} = F_k + F_{k+2}$.

By the inductive hypothesis, we have $L_k = F_{k-1} + F_{k+1}$ and $L_{k-1} = F_{k-2} + F_k$.

To find L_{k+1}, we substitute the expressions from the inductive hypothesis:

$$L_{k+1} = L_k + L_{k-1} = (F_{k-1} + F_{k+1}) + (F_{k-2} + F_k)$$

Simplifying, and using the Fibonacci definition, we get:

$$L_{k+1} = F_{k+1} + (F_{k-1} + F_{k-2}) + F_k = F_{k+1} + F_k + F_k$$

Since $F_{k-1} + F_{k-2} = F_k$, we further simplify to:

$$L_{k+1} = F_{k+1} + F_k + F_k = F_{k+1} + (F_k + F_{k-1}) = F_{k+1} + F_{k+1} = F_k + F_{k+2}$$

This completes the inductive step, showing that if the statement is true for k, it is also true for $k + 1$.

Conclusion:

By the principle of mathematical induction, the statement $L_n = F_{n-1} + F_{n+1}$ holds for all $n > 0$.

3

Pseudocode

Algorithm 1 Recursive Insertion Sort

```
1: function INSERT(A, n)
2:     val ← A[n]
3:     i ← n − 1
4:     while i > 0 & A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← val
9: end function

10: function RECURSIVEINSERTIONSORT(A, n)
11:     if n > 1 then
12:         RECURSIVEINSERTIONSORT(A, n − 1)
13:         INSERT(A, n)
14:     end if
15: end function
```

Worst-Case Running Time

The recurrence for the worst-case running time can be succinctly expressed as:

For $n = 1$, $T(1) = \Theta(1)$, where the array has only one element and is trivially sorted.

For $n > 1$, the recurrence is $T(n) = T(n-1) + \Theta(n)$. In this recurrence, $T(n-1)$ represents the time to recursively sort the subarray of size $n-1$, and $\Theta(n)$ represents the time to insert the last element into its correct position in the sorted subarray, which in the worst case involves comparing it with each of the $n-1$ elements already sorted.

To summarize, the solution to this recurrence is $T(n) = \Theta(n^2)$.

4

a

Given n/k sublists each of length k, the time complexity of insertion sort for each sublist is $\Theta(k^2)$ in the worst case. Since there are n/k such sublists, the total time to sort all of them using insertion sort is:

$$\Theta\left(\frac{n}{k}\right) \cdot \Theta(k^2) = \Theta\left(\frac{n}{k} \cdot k^2\right) = \Theta(nk)$$

b

We need to merge these n/k sorted sublists into a single sorted list. The merge process of merge sort takes $\Theta(n)$ time for two sublists because each element is looked at once. When merging n/k sublists, we do this merging process $\log(n/k)$ times.

So, the time complexity of merging is governed by how many times we can halve the number of lists until we reach 1 list, multiplied by the $\Theta(n)$ work needed for each merge operation. This gives us a total merging time of:

$$\Theta(n \log(n/k))$$

c

We set the modified running time equal to the standard merge sort running time and solve for k:

$$\Theta(nk + n\log(n/k)) = \Theta(n\log n)$$

The optimal k is a sublinear function of n, which grows slower than n but faster than a constant, ensuring the nk term doesn't outweigh the logarithmic component, keeping the overall running time within $\Theta(n\log n)$. So, the largest value of k can be set to $\Theta(\log n)$.

5

A	**B**	**O**	**o**	Ω	ω	Θ
$\lg^k n$	n^ε	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n\sin n$	yes	yes	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n\lg c$	$c\lg n$	yes	yes	no	no	no
$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

6

Complexity

At the first level, the cost is $cn\log n$. At the second level, there are 2 subproblems, each of size $\frac{n}{2}$, and the cost for each is $c\frac{n}{2}\log\frac{n}{2}$. This pattern continues, with each level having twice as many subproblems of half the size, and the cost per subproblem decreasing logarithmically.

The total number of levels is $\log n$ because the subproblem size is halved at each level.

The cost at each level is approximately $cn\log n$, because the log term decreases by 1 for each halving of n, but the number of subproblems doubles.

Therefore, the total cost is the cost per level times the number of levels:

$$\text{Total cost} = cn\log n \times \log n = cn(\log n)^2$$

So the time complexity of the program is $O(n(\log n)^2)$.

Prove

We can use mathematical induction.

Base Case:

For $n = 2$, the recurrence gives us $T(2) \leq 2c$, which is a constant and thus holds for the base case.

Recurrence Tree Levels:

Each level i of the tree has 2^i nodes, each of cost $c\frac{n}{2^i}\log\frac{n}{2^i}$. The total cost at level i is:

$$2^i \cdot c\frac{n}{2^i}\log\frac{n}{2^i} = cn(\log n - i)$$

Summing the Levels:

The tree has $\log n$ levels, so we sum the costs across all levels:

$$\text{Total cost} = cn\log n + cn(\log n - 1) + cn(\log n - 2) + \ldots + cn$$

This forms an arithmetic series in terms of $\log n$, with $\log n$ terms.

Total Cost Calculation:

The sum of the first m terms of an arithmetic series is $\frac{m}{2}(a_1 + a_m)$. Applying this formula, we get the total cost as:

$$\begin{aligned}\text{Total cost} &= cn \times \frac{\log n}{2}(2\log n - (\log n - 1)) \\ &= cn \times \frac{\log n}{2}(\log n + 1) \\ &= \frac{cn}{2}(\log^2 n + \log n)\end{aligned}$$

Since $\log n$ is dominated by $\log^2 n$ as n grows large, the term $\frac{cn}{2}\log n$ can be omitted when considering the big-O notation, yielding:

$$\text{Total cost} = O(n\log^2 n)$$

.

Conclusion:
The total cost of all levels, which represents the running time of the recursive program, is $O(n\log^2 n)$.

7

c

Compare: Algorithm a took approximately 1 minute and 35 seconds to run, reflecting its exponential time complexity due to repeated recalculations. In contrast, algorithm b completed in just 0.032 seconds.

Conclusion: The optimized algorithm (b) is vastly more efficient than the simple recursive approach (a), showcasing the importance of algorithmic optimization and memoization in handling computationally intensive tasks, especially for large input sizes.

d

Using a 4-byte int type in C++ to compute L_{50} in the simple recursive program is not feasible. This is because the int type typically has a maximum value of $2^{31} - 1$ when signed, and Lucas numbers grow exponentially. L_{50} is much larger than this limit, leading to overflow and incorrect results.

The part b) program accurately computes L_{500} using a custom-implemented 'BigInt' class for high-precision arithmetic. This class handles numbers as digit sequences, enabling representation of very large values beyond standard integer limits, thus avoiding overflow and ensuring precise L_{500} computations.