Algorithm Analysis

Group Members and Contributions

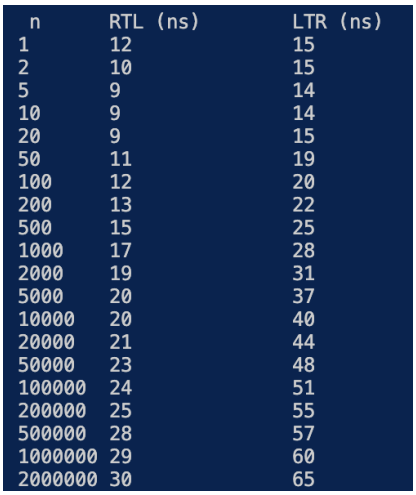
Eric Xu: Wrote the source code for task 2 and the Big O explanation for the code. Also, wrote the search functions for task 3.

Owen Bosticco: Wrote the main code and analysis for task 3. Provided the screenshots for task 2 and 3.

Sam Schlabach: Wrote the code and analysis for the functions in task 1.

Task 1

Screenshots



n	RTL (ns)	LTR (ns)
1	12	15
2	10	15
5	9	14
10	9	14
20	9	15
50	11	19
100	12	20
200	13	22
500	15	25
1000	17	28
2000	19	31
5000	20	37
10000	20	40
20000	21	44
50000	23	48
100000	24	51
200000	25	55
500000	28	57
1000000	29	60
2000000	30	65

Analysis

Both left-to-right and right-to-left methods compute x^n in O(logn) time, and because of their similar qualities, they are widely considered equal. However, in practice and as shown above, the two can differ by quite a substantial amount. The right-to-left method substantially outperformed the left-to-right method when it came to higher powers. This is because RTL only requires one loop to operate and functions mostly with registers rather than using extra memory. However, LTR uses two loops and more memory to compute. Because of this and the structure of the two algorithms, the RTL method is much faster. We didn't encounter any valid values of x and/or n that caused our algorithm to break, but it

could potentially break for extremely large values of x and/or that would cause an overflow error since we had our data types as int.

Task 2

Screenshots

```
Enter positive integer p: 100
Enter base b (2-36): 2
Converted value: 1100100

Enter positive integer p: 1000
Enter base b (2-36): 4
Converted value: 33220

Enter positive integer p: 10000
Enter base b (2-36): 36
Converted value: 7PS
```

Big O Notation

The Big O of our algorithm is O(log(n)). Our algorithm has each iteration of the loop dividing p by b. Therefore, the number of iterations would equal the number of digits in base b. Which would make the time complexity of the algorithm to be $O(\log_b(p))$. And since p is the input (n), this would simplify the Big O notation to be O(log(n))

Task 3

Screenshots

```
Binary Search vs Interpolation Search Benchmark
Trials per size: 20000 (50% hit / 50% miss)

Size      Binary avg (ns)     Interp avg (ns)
-------------------------------------------------
100       41.2                38.9
500       59.7                41.7
1000      66.3                44.8
5000      89.0                47.7
10000     96.2                48.7
50000     119.3               49.7
```

Analysis

Overview:

In this experiment, Binary Search and Interpolation Search were implemented as separate functions and tested on sorted arrays of randomly generated integers. Arrays of increasing size were generated, sorted, and then searched multiple times to measure average runtime. The goal was to understand how each algorithm's performance scaled with increasing array sizes.

Test Setup:

Arrays of sizes 100, 500, 1,000, 5,000, 10,000, and 50,000 were created. Each array was filled with random integers in the range [0, 100,000] and sorted before searching.

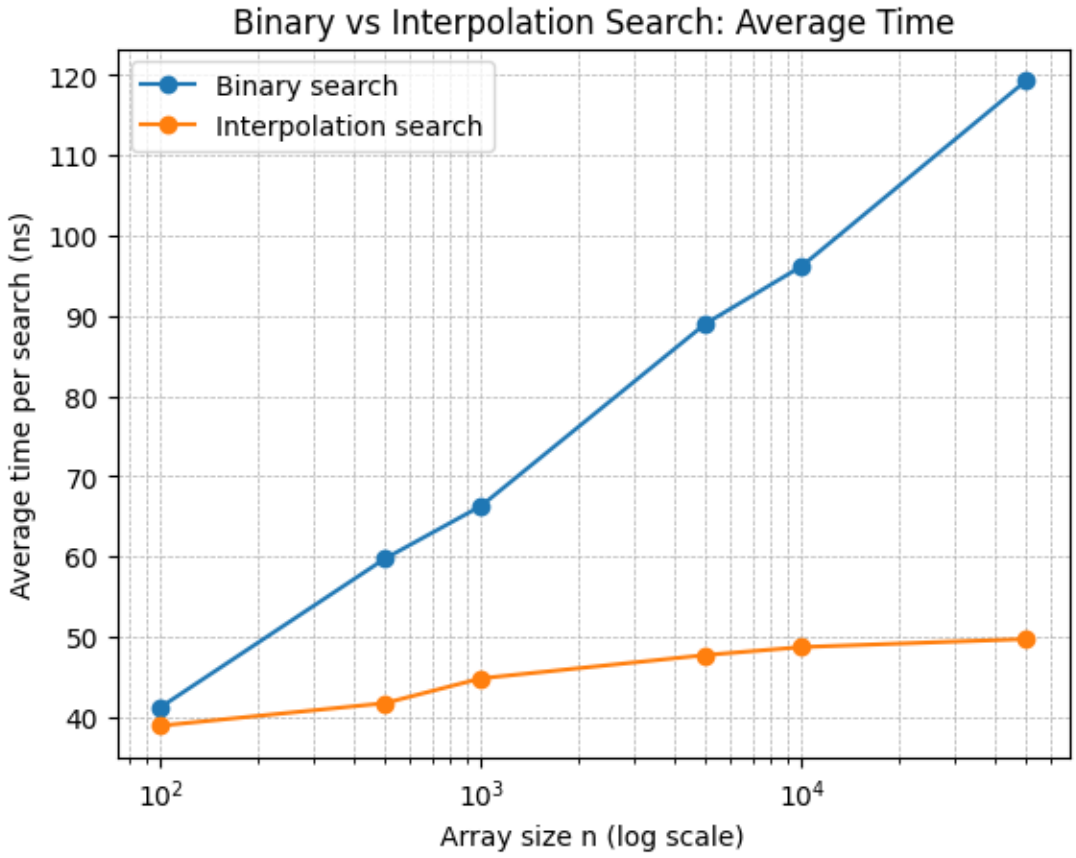
For each array size:

- 20,000 searches were performed
- Around 50% of searches used a value guaranteed to be in the array
- Around 50% used a value guaranteed to be outside the array's range

Average runtime in nanoseconds was computed using std::chrono.

Results:

Binary search runtime increases steadily with array size, while interpolation search remains relatively flat.



Binary Search Behavior

Binary search repeatedly divides the search space in half. This means the number of steps increases with $\log_2(n)$ As the array size grows, more iterations are required, which explains the steady increase in runtime from about 41 ns at size 100 to about 119 ns at size 50,000.

Binary search performance is mostly unaffected by how values are distributed in the array. Whether values are evenly spread or clustered, binary search still performs the same sequence of halving operations. This makes it reliable and predictable, but not always the fastest option.

Interpolation Search Behavior

Interpolation search estimates where the search value should be based on the values at the ends of the array. When data is close to uniformly distributed, this estimate is often accurate, allowing the algorithm to converge in fewer steps.

In this experiment, the arrays were populated with random values over a wide range, which tends to create a reasonably uniform distribution. Additionally, half of the searches were guaranteed misses outside the array's value range. Interpolation search is able to quickly reject these out-of-range values, which significantly reduces average runtime.

As a result, interpolation search remained near constant time across all array sizes, increasing only slightly from about 39 ns to 50 ns.