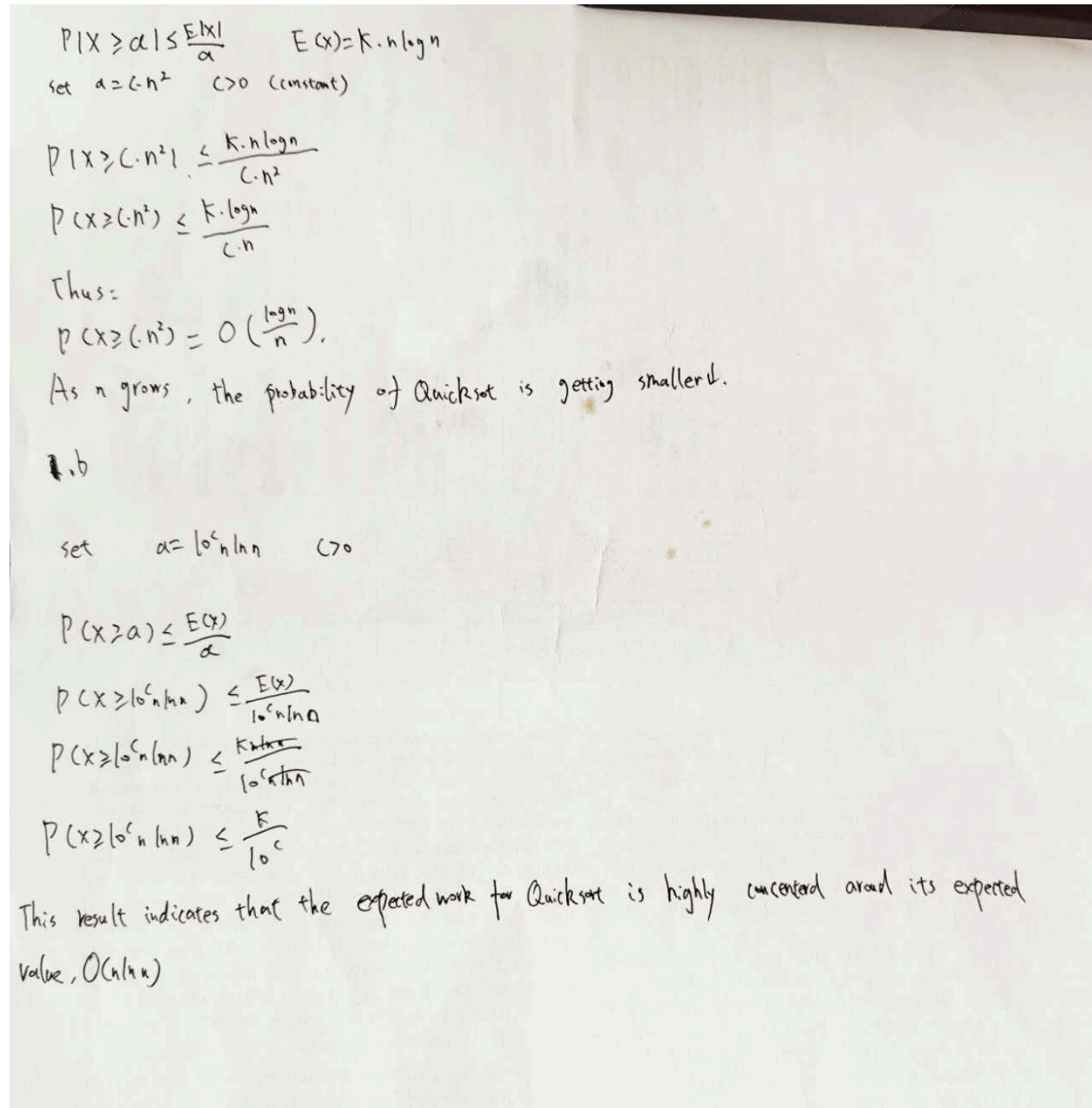


Yan Zhu

1.a



$P(X \geq \alpha) \leq \frac{E(X)}{\alpha} \quad E(X) = K \cdot n \log n$
set $\alpha = C \cdot n^2 \quad C > 0$ (constant)

$$P(X \geq C \cdot n^2) \leq \frac{K \cdot n \log n}{C \cdot n^2}$$
$$P(X \geq C \cdot n^2) \leq \frac{K \cdot \log n}{C \cdot n}$$

Thus:

$$P(X \geq C \cdot n^2) = O\left(\frac{\log n}{n}\right).$$

As n grows, the probability of Quicksort is getting smaller.

1.b

set $\alpha = 10^c \cdot n \ln n \quad C > 0$

$$P(X \geq \alpha) \leq \frac{E(X)}{\alpha}$$
$$P(X \geq 10^c \cdot n \ln n) \leq \frac{E(X)}{10^c \cdot n \ln n}$$
$$P(X \geq 10^c \cdot n \ln n) \leq \frac{K \cdot n \ln n}{10^c \cdot n \ln n}$$
$$P(X \geq 10^c \cdot n \ln n) \leq \frac{K}{10^c}$$

This result indicates that the expected work for Quicksort is highly concentrated around its expected value, $O(n \ln n)$

1b.

Quicksort's expected work, $O(n \ln n)$, is very predictable and rarely deviates significantly. By using Markov's inequality, we can see that the probability of Quicksort performing $10^c n \ln n$

comparisons decreases exponentially as c increases. This means that as the input size grows, the chance of Quicksort taking much longer than its expected time becomes incredibly small.

In practice, this shows that Quicksort is highly reliable, with its performance tightly clustered around the expected $O(n \ln n)$. The algorithm is efficient and rarely hits worst-case scenarios. This “concentration” around the expected work is one of the reasons Quicksort is so widely used—its performance is not just theoretically efficient but also practically consistent for most input sizes. It’s fast, dependable, and unlikely to surprise you with unexpectedly poor performance.

2a.

Algorithm $A'(I, \delta)$; Each run of A has a success probability ϵ ,

$$(1-\epsilon)^K \leq 1-\delta.$$

solving for K :

$$K = \left\lceil \frac{\ln(1-\delta)}{\ln(1-\epsilon)} \right\rceil$$

1. Run A K times

2. After Run, ~~verify~~ verify the result using $C(A(I))$

3. If the $C(A(I))$ confirms ~~is~~ correctness, return result.

4. If K runs fail, probability $1-\delta$

$$\text{work} = O(2W(n) \cdot K) = O(W(n) \cdot \frac{\ln(1-\delta)}{\ln(1-\epsilon)})$$

2b.

1. Run $A(I)$

2. Use $C(A(I))$ to verify the correctness.

3. If $C(A(I))$ confirm correctness, return.

Total work per attempt: $2W(n)$, expected running time $\frac{1}{\epsilon}$.

$$\text{Total work} = O\left(\frac{2W(n)}{\epsilon}\right)$$

- **3a.**

From the table, the running times align well with the theoretical asymptotic bounds:

1. Quicksort with Fixed Pivot: The runtime increases significantly and trends toward $O(n^2)$, especially for imbalanced partitions. This happens because the fixed pivot can lead to poor performance for near-sorted or structured inputs.
2. Quicksort with Random Pivot: The runtime closely matches the average-case complexity of $O(n \log n)$. Random pivoting avoids unbalanced partitions, resulting in more stable and efficient performance on large inputs.
3. Timsort: It performs the best, with runtime consistently near $O(n \log n)$. As Python's built-in sorting algorithm, Timsort is optimized for nearly sorted data, making it highly efficient.
4. Selection Sort: The runtime grows quickly and matches $O(n^2)$. While acceptable for small inputs, it becomes impractical for $n > 5000$.

In conclusion, random-pivot Quicksort and Timsort excel at handling large inputs, while selection sort is only suitable for small datasets.

- **3b.**

Changing the input list type significantly impacts the relative performance of the algorithms:

1. Fixed Pivot Quicksort: For random inputs, it performs decently but is slower than random-pivot Quicksort due to unbalanced partitions. For sorted inputs, its performance degrades to $O(n^2)$, as the fixed pivot causes highly unbalanced partitions, making it inefficient.
2. Random Pivot Quicksort: It handles both random and sorted inputs consistently with $O(n \log n)$ performance. The random pivot avoids unbalanced partitions, making it resilient to sorted inputs and maintaining efficiency.
3. Timsort: It performs well on random inputs with $O(n \log n)$ complexity but excels on sorted or nearly sorted inputs, often achieving $O(n)$ due to its optimization for ordered data.
4. Selection Sort: The performance remains $O(n^2)$ regardless of input type, as it does not exploit input structure. It is slow and inefficient for large lists.

n	qsort-fixed-pivot	qsort-random-pivot	timsort	ssort
100	0.197	0.251	0.013	0.227
200	0.364	0.492	0.014	0.774
500	1.352	1.514	0.044	5.004
1000	1.536	2.706	0.107	16.546
2000	3.392	4.339	0.209	40.193
5000	5.220	6.898	0.369	220.267
10000	10.896	14.346	0.793	873.619
20000	22.646	29.419	1.548	3534.808
50000	61.121	77.508	4.543	22035.028
100000	132.558	165.442	9.934	88507.496