Lab 04

Phone and Ladder

Part A: One Phone

If you only had one phone, then you would need to test every rung, starting from the bottom. That way, when the phone breaks, you know exactly which rung it broke on, and you don't need any more tests. The time complexity for one phone would therefore be $O(n)$.

Part B: Two Phones

The generic strategy would be to use the first phone to get a rough estimate of where the breaking point is, then going through that rough estimate zone with the second phone using the linear algorithm to find the exact location. So you would drop the first phone at rung p, then $2p$, then $3p$, all the way until rung n. Then you would have an interval of length p where you know the phone's breaking point must be, and you can search those rungs one by one. The optimal value for p is the one where each phone, on average, gets the same number of drops. This is because the first phone will be dropped $\frac{n}{2p}$ times on average, and the second phone will be dropped $\frac{p}{2}$ times on average. So we minimize $\frac{n}{2p} + \frac{p}{2}$:

$$\begin{aligned} T(n) &= \frac{n}{2p} + \frac{p}{2} \propto \frac{n}{p} + p \\ \frac{\delta T}{\delta p} &= 1 - \frac{n}{p^2} = 0 \\ \frac{n}{p^2} &= 1 \\ p^2 &= n \\ p &= \sqrt{n} \end{aligned}$$

We see that the best-case runtime is when $p = \sqrt{n}$, where both phones get dropped p times. The average runtime is therefore $O(\sqrt{n})$.
For a ladder with 25 rungs we would drop the first phone at heights $\{5, 10, 15, 20\}$, and then at most 4 tests to determine the exact rung. This gives a maximum of 9 trials, a minimum of 5, and an average of 5.5. Here are those numbers:

Rungs (n)	p	worst case	best case	average
25	5	9	2	5.5
50	7	13	2	14.5
100	10	19	2	9.5

Stability

1. Insertion sort: **Yes**. Sorting is determined by a strict comparison, so no equal elements are swapped. This means original order is preserved.
2. Merge sort: **Yes**. When merging, we put the element from the left array into the merged one if `L[i]<=R[i]`. The `<=` means that equal elements will be inserted with the left ones first, then the right, preserving the original order.
3. Bubble sort: **idk**. I couldn't find bubble sort in the book. Assuming you only swap elements when they are strictly in the wrong order, bubble sort is stable.
4. Quicksort: **No**. It checks whether to move the items with <=, so it might reorder some equal elements.
5. Counting sort: **yes**. It's used in radix sort ;). Elements of the same value get treated the same, so they never get reordered.
6. Radix sort: **Yes**. It's based on counting sort, which is stable.
7. Bucket sort: **Yes**. Nothing is reordered when placed into buckets.

Linear Time Sorting Restrictions

1. What happens with counting sort if the range of the values in A is arbitrarily large, e.g. the n values in A are between 0 and n2?
   Counting sort would break down. You could change the algorithm to make it not fail, but then it would require arbitrarily large amounts of memory.
2. What happens with radix sort if the number of digits isn’t constant, e.g. the n values in A have n-digits?
   You would have to pad each number with enough preceeding zeros to make all the numbers match. Otherwise the algorithm would not be comparing digits with equal values (ie, it could be comparing one number's ones place with another's 10's place)
3. What happens with bucket sort if the values in A are not uniformly distributed over $[0, 1)$, e.g. the values in A are skewed so most are in the interval $[0, 1/n)$?
   It will no longer be $O(n)$, because there will no longer be only a few numbers in each bucket. This will leave a lot more work for insertion sort. The values' distribution must be the same as the buckets'.

Statistic

1. The worst-case runtime is $O(n^2)$, since the partition is the first element.
2. If j = 1 or j = n, then there is a Θ(n)-algorithm to find those statistics. In your Google doc, pseudocode this algorithm for the case when j = 1 and briefly justify the worst-case runtime.

```
def first_order_statistic(A):
        minimum = A[0]
        for i in A:
                if i<minimum:
                        minimum=i
        return minimum

def nth_order_statistic(A):
        maximum = A[0]
        for i in A:
                if i>maximum:
                        maximum=i
        return maximum
```

The worst-case runtime is $O(n)$. The worst case occurs when the max/min is at the end of the list, in which case we have to check every element in the list once.

3. How can you modify your generic statistic algorithm to also achieve an expected linear runtime? Explain. (Hint: think about modifications made to quicksort to improve its runtime.)
   I think it already is linear time. Each level of recursion uses (in the average case) half the time of the previous level, because we only do a recursive call on the half where we know our target value to be. Recursion ends when we've halved it so much that we're only looking at one element. As $n \to \infty$, this just means forever, so the runtime is:

$$\begin{aligned} T(n) &= n + \frac{1}{2}n + \frac{1}{4}n + \cdots + 1 \\ &= n \sum_{i=1}^{\infty} 2^{(-i)} \\ &= 2n \\ &= O(n) \end{aligned}$$