Linear-Time Sorting

1. (a) `[byte, pits, bits, pins]`
   (b) `[pins, byte, pits, bits]`
   (c) `[pins, pits, bits, byte]`
   (d) `[bits, byte, pins, pits]`

2. (a) To output the sorted array, we need to know how many occurrences there are of each value $1, 2, \ldots, k$. So for each of these we do a binary search on the number of occurrences: for example, we ask "Are there $n/2$ or more 1's in the array?" and so on until we determine the number of occurrences for each of $1, 2, \ldots, k$. Each binary search takes $O(\log(n))$ time and we do k searches, so in total this takes $O(k \log(n))$ time.

   The pseudocode is as follows:

```
def binary_search(value, low, high):
  if low >= high:
    return low
  mid = (low + high) / 2
  answer = ask("Are there more than `mid' occurences of `value' in the array?")
  if answer is "yes":
    return binary_search(value, mid + 1, high)
  else:
    return binary_search(value, low, mid)

def main():
  counts = []
  for i = 1 to k:
    counts.append(binary_search(i, 0, n))
  for i, count in enumerate(counts):
    for j = 1 to count:
      print i
```

   (b) The above assumes knowledge of n and k. If you did not know what these values were, you could also do a binary search for them. For example, you could ask "Is $n < c$?" for $c = 2, 2^2, 2^3, \ldots$. If the answer is first "YES" for $c = 2^b$, then we binary search for n between 2^{b-1} and 2^b. This will take in total $O(\log n)$ time: $O(\log n)$ to find the range we need to binary search, and $O(\log n)$ to do the binary search. This same procedure can be used to find k.

   (c) An algorithm that asks at most c questions can have at most 2^c outputs because each sequence of answers can correspond to only one output. For any series of c yes/no questions, there are 2^c sequences of possible answers. So if there are N distinct sorted arrays (for given values of n and k) then any algorithm must ask at least $\log N$ questions in the worst case.

   A lower bound on N, the number of distinct sorted arrays of length n containing the elements $\{1, \ldots, k\}$, is given as follows: for each value $1, \ldots, k-1$ let it occur anywhere between 1 and n/k times. Let the value k occur the number of times that will fill out the array to n numbers. This

results in $\left(\frac{n}{k}\right)^{k-1}$ distinct sorted arrays. Thus we have a lower-bound on the worst-case number of questions: $\log\left(\frac{n}{k}\right)^{k-1} = (k-1)\log\frac{n}{k} = \Omega(k\log\frac{n}{k})$.

Alternative way to count N ("stars and bars" approach): the number of distinct sorted arrays is given by the number of ways to distribute n balls into k boxes, which is $\binom{n+k-1}{k-1} = \frac{(n+k-1)(n+k-2)\cdots(n+1)}{(k-1)!} \geq \frac{n^{k-1}}{k^{k-1}} = \left(\frac{n}{k}\right)^{k-1}$. This simplification follows from the fact that $(n+k-1)(n+k-2)\cdots(n+1) > n^{k-1}$ and $(k-1)! < k^{k-1}$.

A thought process you could have used to arrive at this algorithm asks why is it sufficient for this problem to lower-bound the number of ordered arrays, instead of counting exactly? Once you have understood this, use a counting argument: how can you lower-bound the number of ordered arrays are there that consist of n integers $\{1,\ldots,k\}$ (not necessarily distinct)? There are a number of ways to do this; we suggest you do NOT use Stirling's approximation: you don't need this in order to prove the result, and it will be complicated.

Randomized Algorithms

---

1. (a) Let X be a random variable that represents the runtime of the randomized algorithm. The problem statement tells us $E[X] = 1$. Then:

$$Pr[X \geq 10] = Pr[X \geq 10E[X]] \leq 1/10$$

   (b) We assume successive executions of `BigProblem1()` are independent. Let X be the random variable representing the runtime of `BigProblem1()`. Again, $E[X] = 1$. Then:

$$\begin{aligned} Pr[\text{no output after 10 seconds}] &= Pr[X \geq 8]Pr[X \geq 2] \\ &= Pr[X \geq 8E[X]]Pr[X \geq 2E[X]] \\ &\leq \frac{1}{8} \cdot \frac{1}{2} \\ &= 1/16 \end{aligned}$$

2. (a) In English, our algorithm does the following:

   - For each pair of points (x_i, y_i) and (x_j, y_j), compute the slope and intercept of the pair.
   - Sort the pairs by according to (m, b).
   - Find the longest consecutive sequence of identical (m, b) in the sorted list of pairs.
   - Return the list of points in this longest consecutive sequence.

   There are $O(n^2)$ pairs of points. For each pair of points (x_i, y_i) and (x_j, y_j), we can compute the slope and intercept of the pair in $O(1)$-time. Since we can compare each (m, b) in $O(1)$-time, sorting the n^2 pairs of (m, b) requires $O(n^2 \log n^2) = O(n^2 \log n)$-time.

   (b) In English, our algorithm does the following:

   - Sample two points uniformly at random and compute (m, b).
   - For all other points, count how many satisfy $y = mx + b$ and maintain a set of these points.
   - If the total number of points on this line is n/k, return this set of points.

   Modeling the expected value of t trials until a single success of both points being chosen from the correct maximum cardinality set is equivalent to modeling a geometric distribution. The probability of choosing two points in the set is $\frac{n}{k}/n \cdot \frac{n}{k}/n = \frac{1}{k^2}$. Thus, the expected value of t, by the geometric distribution, is k^2. Additionally, for each trial, the algorithm loops through all other $n - 2$ points in A, and performs constant work (calculating a slope) on each point. Thus, the expected runtime of this algorithm is $O(k^2 \cdot n)$, or $O(n)$.

   (c) $O(\infty)$. The algorithm is not guaranteed to terminate, since it does not guarantee picking a pair of points in the maximum cardinality set

Hash Functions

---

1. (a) 26^8
   (b) $n^{(26^8)}$
   (c) $26^8 \log(n)$
   (d) Yes.
   (e) 1
   (f) $1/n$
   (g) $1/n$

2. (a) **Style note:** Here are a few acceptable ways of writing the proof.

   Soln. 1 We can decompose the probability into the sum of conditional probabilities based on the value of $h(x_i)$; that is, $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^{n} \Pr[h(x_i) = k] \cdot \Pr[h(x_i) = h(x_j) | h(x_i) = k]$. But what is $\Pr[h(x_i) = h(x_j) | h(x_i) = k]$ exactly? It is in fact just $\Pr[h(x_j) = k]$, since once we know that $h(x_i) = k$, the only way for $h(x_j)$ to equal $h(x_i)$ is if $h(x_j) = k$. Then $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^{n} \Pr[h(x_i) = k] \cdot \Pr[h(x_j) = k]$.
   Now, because $\mathcal{H}$ consists of every possible function from $\mathcal{U}$ to $\{1, ..., n\}$, we know that there are equally many functions mapping x_i to every possible value in $\{1, ..., n\}$. Then $\Pr[h(x_i) = k]$ is exactly $1/n$, and similarly for $\Pr[h(x_j) = k]$. This tells us that $\Pr[h(x_i) = h(y_i)] = \sum_{k=1}^{n} \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n}$, since we are summing $(\frac{1}{n})^2$ up n times. This proves that indeed the probability of collision is indeed exactly $1/n$, as desired.

   Soln. 2 The number of hash functions such that $h(x_i) = h(x_j) = k$ is n^{M-2} for each $k = 1, \ldots, n$, because there are $M - 2$ remaining elements of $\mathcal{U}$ that can be assigned. So the total number of hash functions such that $h(x_i) = h(x_j)$ is exactly n^{M-1}. Thus the probability that $h(x_i) = h(x_j)$ is exactly

   $$\frac{n^{M-1}}{|\mathcal{H}|} = \frac{n^{M-1}}{n^M} = \frac{1}{n}$$

   , as desired.

   (b) One way to see that this is the case is to notice that, by removing h_1 from the set of possible functions, we are making sure that every pair of elements is strictly less likely to collide; that is, in part (a), the probability we got was based on the existence of this function in the set, and removing the function must make the probability of collision strictly lower.

   We can prove this more formally. Fix $x_i, x_j \in \mathcal{U}$. We will count the number of $h \in \mathcal{H}$ that have $h(x_i) = h(x_j)$.

   First, observe that for all $k \in \{2, \ldots, n\}$, the number of h such that $h(x_i) = h(x_j) = k$ is exactly n^{M-2}; that is, for each of the remaining $M - 2$ items of $\mathcal{U}$, we have n choices each. The fact that $\mathcal{H}'$ is missing h_1 does not affect this computation, since $h(x_i) = k \neq 1$, so $h \neq h_1$ in this case.

   On the other hand, for $k = 1$, the number of h such that $h(x_i) = h(x_j) = 1$ is exactly $n^{M-2} - 1$. The reasoning is exactly the same as before, except we throw h_1 out of the count.

Then, the probability

$$\begin{aligned}
\Pr h(x_i) = h(x_j) &= \frac{\text{number of } h \in \mathcal{H}' \text{ so that } h(x_i) = h(x_j)}{|\mathcal{H}'|} \\
&= \frac{n^{M-2} - 1}{n^M - 1} + \sum_{k \neq 1} \frac{n^{M-2}}{n^M - 1} \\
&= \frac{n^{M-1} - 1}{n^M - 1} \\
&< \frac{1}{n}.
\end{aligned}$$