CS561, HW3

Ryan Scherbarth, University of New Mexico

September 2024

1. You are on an island where each inhabitant is either a knight or a knave. When asked a question, knights always tell the truth, but knaves may either lie or tell the truth. On this island, you can only ask questions of the form: "Hey Person X: what is Person Y's type?". Each person always answers a question about someone else in the same way, so there's no reason to ask the same question twice.

   - (a) Assume there are n people on the island, and a strict majority of them are knights. Describe an efficient algorithm to identify the type of each person.
     Hint 1: You can do this in $o(n^2)$ questions.
     Hint 2: First try to identify 1 knight. Use recursion!

X	**Y**	**Result**	**Inference**
Knight	Knight	Both said Knight	Both are the same
Knave	Knave	Both said Knight	Both are the same
Knight	Knave	Both said Knave	Both are different
Knave	Knight	Both said Knave	Both are different

There are n total on the island
There are at least $\frac{n}{2} + 1$ Knights
Therefore, there are n - Knights number of Knaves.

We can start by grouping all of the individuals evenly, and ask each group containing X and Y what the other is. For any group where X and Y agree in their answers, and both respond with "Knight", we will put the group aside.
For any group where X and Y agree in their answers, and both respond with "Knave", we will keep this group in the recursive pool and continue searching.

Case 1: n is odd
The fact that we have a strict majority of knights assures us that once we reach the bottom of our recursive pool, we will be left with a single knight.

Case 2: n is even
Once we reach the bottom of our recursive case, we re-shuffle and continue. Due to the fact that we have a strict majority of knights, Once we make it to our last group we can know for sure that both will be knights. Again, we will now loop over our unverified groups asking the knight what each of them are.

In the worst case, we cut our recursive pool in half, setting our new group size to $\frac{n}{2}$. Despite setting a number of groups aside that we will need to re-evaluate later, this means our running time is $O(\log(n))$. Since we are setting a random sub portion of groups aside to be evaluated later, we can ignore this arbitrary constant number of groups, which will be absorbed by the $\log(n)$ term.

Therefore, our algorithm is bounded by $o(n^2)$ with a running time of $O(\log(n))$.

- (b) Prove that if at most half of the inhabitants are knights, then it is impossible to solve the problem.

Our solution for part a entirely relies on the foundation of $Knights \geq \frac{n}{2}$. That way, we can continue shuffling and singling out a single group that we will then know contains a knight because of their consistent answers.

This idea no longer is possible if we don't have a strict majority, which will remove both of our possible base cases. Without a strict majority, we are no longer guaranteed to reach the bottom of our recursive method with a group that contains someone who is determined to have been telling the truth.

In this case, we just end up often placing every single one of the groups aside (removing from the recursion pool) and then have no way to solve.

2. A product of matrices is fully parenthesized if it is either a single matrix, or the product of two fully parenthesized matrix products, surrounded by parenthesis. For example, $(A_1((A_2A_3)A_4))$ is fully parenthesized. Prove by induction that any full parenthesization of a product of n matrices has exactly $n-1$ pairs of parenthesis.

   Base case: $n = 1$
   when there is only one matrix, no multiplication is required, therefore no parentheses are needed.
   $1 - 1 = 0$,
   so the base case holds true.

   Let $j > n$

   Inductive Hypothesis:
   Assume that for any number of matrices, m, where $1 \leq m \leq n$, the number of parentheses required for the full parenthesization of m matrices is $m-1$.

   Inductive Step:

$$\begin{aligned} f(n+1) &= (i-1) + ((n+1-i)-1) + 1 \\ &= i - 1 + n - i + 1 \\ &= n \end{aligned}$$

   Here, we have shown that the number of parenthesis, $f(n)$, gives us n parenthesis for an input size of $n+1$.

3. Consider a rooted binary tree with nodes are labelled as follows. The root node is labelled with the empty string. Then, any node that is a left child of a node with name o. receives the name oL and any node that is the right child of that node receives the name oR.
Give a recurrence relation returning the number of R's in all labels of all nodes. For example, the following tree has 10 R's.
Base case: If v is a leaf node (it has no children), then $f(v) = 0$, since tehere are no further nodes.
Recursive Case: If v is not a leaf node, let $f(v)$ be recursively defined in terms of the values for it's left child $l(v)$ and it's right child $r(v)$.
let $l(v)$ be the left child of v, and $r(v)$ be the right child of v.
By the definition of the problem, $r(v)$ is the right child, the number of r's in the subtree rooted at $r(v) = f(r(v)) + s(r(v))$ where we let $s(r(v))$ represent the number of nodes in the subtree rooted at $r(v)$.
we see that every node in the subtree of $r(v)$ inherits one additional r from the right child relationship.
Therefore, we have the equation;

$$f(v) = f(l(v)) + f(r(v)) + s(r(v))$$

4. In this problem, you will use the following facts: (1) any integer can be uniquely factored into primes; (2) the number of primes less than any number m is $\Theta(\frac{m}{\log(m)}$ (This is the prime number theorem).
We will also make use of the following notation for integers x and y (1) $x|y$ means that x "divides" y, which means that there is no remainder when you divide y by x. and (2) $x \equiv y \pmod{p}$

   (a) Show that for any positive integer x, x factors into at most $\log(x)$ unique primes

$$x = p_1 * p_2 * p_3 * \cdots * p_k$$
$$x \geq 2 * 3 * 5 * \cdots * p_k$$

Since the product of the first k primes grows very quickly, the product is greater than 2^k because each prime $p_i \leq 2$.
therefore,

$$x \leq 2^k$$
$$\log(k) \geq k \leq (2)$$
$$k \leq \frac{\log(x)}{\log(2)}$$

   (b) Use prime number theorem to show $p|x = O(\frac{(\log(x))(\log(m))}{m})$

The prime number theorem says a number of primes less than or equal to m is approx. $f(m) = \Theta(\frac{m}{\log(m)})$ where $f(m)$ is the prime counting function.
The probability p divides x is given by $P(p|x)$, where;

$$x = p_1^{e_1} * p_2^{e^2} * \cdots * p_k^{e_k}$$

the total number of primes is less than or equal to $\frac{m}{\log(m)}$, so the probability that a uniformly random prime p, chosen from the primes less than or equal to m, divides x is;

$$P(p|x) = \frac{\text{prime dividing } x}{\text{total primes } \leq m} \leq \frac{\log(x)}{f(m)} = \frac{\log(x)}{\frac{m}{\log(m)}}$$
$$P(p|x) \leq \frac{\log(x)}{\frac{m}{\log(m)}} = \frac{\log(x)(\log(m)}{m}$$
$$P(p|x) = O(\frac{\log(x)\log(m)}{m})$$

Therefore the probability that a prime p uniformly chosen at random from all primes less than or equal to m divides a given int x is $O(\frac{\log(x)\log(m)}{m})$.

(c) Let x, $y < n$, s.t. $x \neq y$. Prove that $x \equiv y(mod(p))$ is $O(\frac{\log(n)\log(m)}{m})$

An equivalent statement to $x \equiv y(mod(p))$ is saying p divides the difference; $p|(x-y)$.
The probability that a random prime $m \leq m$ divides any integer z is $O(\frac{\log(z)\log(m)}{m})$
Since x and y are both positive and less than n, we know that $z = x - y$ is at most $n - 1$.

Using part (c) we can get the probability of a randomly chosen prime $p \leq m$ divides $x - y$ is at most $O(\frac{\log(x)\log(m)}{m})$. We can then let $z \leq n - 1$;

$$P(p|(x-y)) \leq O(\frac{\log(n)\log(m)}{m})$$

Therefore, we have proven the probability that $x \equiv y(mod(p))$ is $O(\frac{\log(n)\log(m)}{m})$.

(d) if $m = \log(n)^2$ then what is the probability that $x \equiv y(mod(p))$

We can start by substituting $m = \log_2(n)$;

$$\begin{aligned} m &= \log_2(n) \\ &= \log(\log_2(n)) \\ &= \log\log_2(n) \\ &= \log\log(n) - \log\log(2) \end{aligned}$$

Therefore for large n we can say $\log(m) \leq \log\log(n)$.

$$\begin{aligned} P(x \equiv y(mod(p)) &= O(\frac{\log(n)(\log\log(n))}{\log_2(n)}) \\ &= \frac{\log(n)(\log\log(n))}{\frac{\log(n)}{\log(2)}} \\ &= O((\log\log(n))(\log(2))) \\ &= O(\log\log(n)) \end{aligned}$$

As n gets large, $\log\log(n)$, representing the probability, is slowly increasing.

(e) Apply result to a new problem.

In our algorithm we start by choosing a random prime p. It is chosen uniformly at random from the set of primes less than or equal to some arbitrary upper bound m. We calculated the number of primes less than or equal to the current in part a with $f(x)$.

Next, we compute the residuals; $xmodp$ and $ymodp$, and they send each other these computed residues.

If $xmodp \neq ymodp$, then we we know $x \neq y$.
If $xmodp = ymodp$ then it would be possible for $x = y$.

The error occurs when $x \neq y$ but $x \equiv ymodp$ for some p. If this is the case, then p divides the difference $p|(x - y)$ as we computed in a previous part.
While the rate of error is very low as we increase p, we can still expect that case to be possible and just restart the process when we hit the error case.