Rowan Lochrin
CSC445 - Alon Efrat
2/15/18
Homework 1

1. First note that the recursive structure of the algorithm means that the runtime of the function for $n$ is equivalent to the runtime of calling the function without the recursive call.

   Also notice that the **if** statement is true for 1 out of 19 values of $n$, meaning we can consider it running on every value of $n$ without changing the order of the runtime of the algorithm. As the runtime of the algorithm without the **if** statement is some constant multiple of the runtime of the algorithm with it.

   The print statements runs $n\sqrt{n}$ times for every value of $n$ so the total runtime of our algorithm is
   $$1 + 2\sqrt{2} + 3\sqrt{3} + ... + n\sqrt{n}$$
   and
   $$\frac{n^2\sqrt{n}}{4} < 1 + 2\sqrt{2} + 3\sqrt{3} + ... + n\sqrt{n} < n^2\sqrt{n}$$
   meaning this algorithm is on the order of $O(n^2\sqrt{n})$ and $\Omega(n^2\sqrt{n})$ this implies that it has a tight bound $\Theta(n^2\sqrt{n})$.

2. $$\begin{aligned}\log_2(n!) &= \log_2(n * (n-1) * (n-2) * ... * 1) \\ &= \log_2(n) + \log_2(n-1) + ... + \log_2(1) \\ &= \log(2)(\log(n) + \log(n-1) + ... + \log(1))\end{aligned}$$

   Discarding the constant we can see that
   $$\frac{n\log(n)}{2} < \log(n) + \log(n-1) + ... + \log(1) < n\log(n)$$
   So we have an upper and lower bound for $\log_2(n!)$ [1] that are both constant multiples of $n\log(n)$ we know $log_2(n!) = \Omega(n\log(n))$

3. I would use a hash table with open addressing. Every element in our hash table will be a $3 \times 3$ array that represents the board.

   (a) **Size** We need to know how many slots will be in our table. There are 6046 possible legal board positions in tic-tac-toe and for performance reasons it would be nice if the size of our table was power of two, So our table will have $2^{13} = 8192$ rows.
   This means that even if we encounter every possible board position our data structure will not be full. However the empty cells will reduce the number of collisions, speeding up insertion, deletion and searching.

   **Hashing Function** First we need a way to translate every board position into a unique space on the table. We will create a function for this. We will first map every square to a prime number $p_1, ..., p_9$ then raise for every $p_i$

[1] Technically we've found bounds of $\log(n!)$ after discarding that constant earlier but constants don't impact the order so it won't change our result.

- If square $i$ is blank $p_i \to p_i^0$
- If square $i$ is an X, $p_i \to p_i^1$
- If square $i$ is an O, $p_i \to p_i^2$

Then take the product of all the cells modulo the size of the table $-1$ [2] e.g.

$$h\left(\begin{array}{c|c|c} \text{X} & & \\ \hline \text{O} & \text{O} & \\ \hline & & \text{X} \end{array}\right) \to \begin{array}{c|c|c} 3^2 & 5^0 & 7^0 \\ \hline 11^1 & 13^1 & 17^0 \\ \hline 19^0 & 23^0 & 29^2 \end{array} \to 3^2 11^1 13^1 29^2 \quad \mod 2^{13}-1$$

$$\begin{aligned} &= 1082367 \quad && \mod 2^{13}-1 \\ &= 1155 && \mod 2^{13}-1 \end{aligned}$$

Note that when we take the product we never have to consider numbers larger then the modulus so this step is quick even for fully populated boards.

**Probing function** We will use double hashing to resolve collisions. We can chose a different set of primes to map to spaces on our board for our second hash function such that no prime is shared with the primes in the original function.

(b) For strategic purposes, flipping or rotating the board has no impact on the information in the position. In this way we can reduce the number of positions we need to consider by factoring out symmetrical positions. When we insert or search the table we will first transform that target or insertion board to it's cardinal form.
You can think of what we hope to accomplish here like making a rule that tells us which way a tic-tac-toe game is supposed to be rotated (or flipped) from it's contents. For our purposes the cardinal form of a board is the rotation or flip that that gives us the greatest integer when it's run through a valuation function $v$. Then whenever we try to insert or search for a board in our database we will first transform that board to it's cardinal form.
We will define $v$ like we did in the last question but make sure to chose primes such that no primes are used in both functions. This way the cardinal positions of the boards are evenly distributed across the table (e.g. If we used the same function for $v$ as we did for $h$ the cardinal position of a board would always have the highest under hash and entries would concentrate around the end of the table). Also we should make sure for any board there are no ties for the highest valuation under $v$, as that would give us a board with an ambiguous cardinal position for that board.
If we define $v$ like we did $h$ but without the final modulo we can meet both of these conditions.
There are 8 symmetries we will consider.

[2]This means that the last cell will always be blank but 8191 is prime which should give us a more even distribution

| | |
|---|---|
| $I$ | Identity (no rotation or flip) |
| $R_{90}$ | Rotation by 90 degrees |
| $R_{180}$ | Rotation by 180 degrees |
| $R_{270}$ | Rotation by 270 degrees |
| $H$ | Flip along the horizontal axis |
| $V$ | Flip along the vertical axis |
| $D$ | Flip along the primary diagonal |
| $D'$ | Flip along the secondary diagonal |

So for a board with squares numbered one through nine some examples of symmetries are

$$I = \begin{array}{c|c|c} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \end{array} \quad R_{90} = \begin{array}{c|c|c} 7 & 4 & 1 \\ \hline 8 & 5 & 2 \\ \hline 9 & 6 & 3 \end{array} \quad H = \begin{array}{c|c|c} 7 & 8 & 9 \\ \hline 4 & 5 & 6 \\ \hline 1 & 2 & 3 \end{array}$$

To demonstrate this, let $B_S$ be a board under a particular symmetry $S$. Then suppose

$$B_I = \begin{array}{c|c|c} \text{X} & & \\ \hline & \text{X} & \\ \hline & & \text{O} \end{array}$$

If we have another board

$$C_I = \begin{array}{c|c|c} \text{O} & & \\ \hline & \text{X} & \\ \hline & & \text{X} \end{array}$$

Notice how the set of all symmetries of $B_I$, and the set of all symmetries of $C_I$ contain the same boards,

$$\begin{aligned} &\{B_I, B_{R_{90}}, B_{R_{180}}, B_{R_{270}}, B_V, B_H, B_D, B_{D'}\} \\ &= \{C_{D'}, C_V, C_D, C_H, C_{R_{90}}, C_{R_{270}}, C_{R_{180}}, C_I\} \\ &= \{C_I, C_{R_{90}}, C_{R_{180}}, C_{R_{270}}, C_V, C_H, C_D, C_{D'}\} \end{aligned}$$

So the maximum of both sets under our valuation function $v$ must be the same so both $C$ and $B$ will have the same cardinal position. e.g. If $B_I$ is the cardinal position of $B$,

$$B_I = C_{D'}$$

If $I$ is greater than any other transform of $B$, so $D'$ must be greater then any other transformation of $C$. This means it also must map to the same cardinal position.
This will reduce the size of our table by a factor of 8 so our table will have 1024 entrys.

(c) The only modification you'd have to make to get this to work with a chess game is you would map every piece (e.g. black knight, white pawn, etc) to an exponent and every game square to a different prime like we did for tic-tac-toe.
This algorithm could be simplified by only considering a few squares and not the whole board. This would mean that many positions would hash to the

same table value, but unlike tic-tac-toe there are more possible chess positions then atoms in the universe. This means that any hash table you create will have a lot of overlapping hashes anyway.

4. We can see that the outer **while** statement iterates through the whole list so the inner **while** loop will run once for every node. Because this loop will run through the remainder in the list for every node $N_k$ it acts on, it will have a runtime of $n - k$. So the total runtime of our program is

$$f(n) = \sum_{k=0}^{n}(n-k) = \frac{1}{2}(n+1)n$$

and because

$$\frac{1}{2}n^2 < f(n) < n^2$$

We know that $O(f(x)) = n^2$ and $\Omega(f(x)) = n^2$ implying $\Theta(f(x)) = n^2$.

5. (a) The worst case running time for an insertion is when a new array has to be allocated and the elements from the old array copied into it. Because the array scales by a factor of 2 every time, this step will always happen after input $2^n + 1$ (or $2n$ counting the zeroth element as one insertion). So for the worst case scenario we will assume $|A| = 2^n$ for some $n$. Meaning that we will have to allocate an array of size $2^n$ and copy over $2^{n-1}$ elements into it. Memory allocation is non-deterministic so we can't analyze the runtime of array allocation[3]. We know that all the elements that were in $A$ at the previous step must be copied over over, so our runtime is $2^n - 1 = |A|/2$.

   (b) The worst case running time for a sequence of inserts is when the last insertion triggers a size change so $|A| = 2^n$ for some $n$. Now because we are dealing with sequences of inserts we also have to consider all the previous inserts. Any insert that does not change the size of an array has a constant runtime. We can assume there will be approximately $|A|/2$ of these inserts. Any insert that does change the size of an array will fail when the runtime is equal to the number of elements inserted so far, and will fail on elements. These inserts happen on powers of 2 so we know the total runtime for these types of inserts will be

$$2^1 + 2^2 .... + 2^{n-1} = 2^n - 1 = |A| - 1$$

   So the total runtime of all insertions is

$$\frac{3}{2}|A| - 1$$

   about triple the runtime for just the last element.

   (c) Because the array starts at size 1 and we multiply the size of the initial array by $(1 + \alpha)$ whenever we go up in size, the size of the array must always be

$$1(1+\alpha)(1+\alpha)...(1+\alpha) = (1+\alpha)^m$$

[3]If we are allocating memory with a command like *calloc* that 0's every cell of the new array this step would take $2^n$ operations

For some $m$. If we insert $n$ elements the size of the array will be $(1+\alpha)^m$ for the smallest integer $m$ such that $(1+\alpha)^m \geq n$. So $m = \lceil \log_{(1+\alpha)} n \rceil$ and we'll have to sum up the runtime of all increases in size along the way, remembering for runtime purposes since we are only moving a number of elements equal to the previous size of the array, we only need to sum up to the previous reallocation. $\lceil \log_{(1+\alpha)} n \rceil - 1$. Giving us a final runtime of

$$\sum_{i=0}^{\lceil \log_{(1+\alpha)} n \rceil - 1} (1+\alpha)^i$$

Which is on the order of $O(n)$.

(d) Both methods are on the same order but the $m^2$ method will be slightly faster if declaring arrays is free. This is because on expansion of the array you will need to copy over all elements in the previous one. The $m^2$ will most likely have fewer expansions of the array and will need to expand less once the array is bigger. To be precise, once $m > (\alpha + 1)$ there will be fewer expansions. However the downside of the $m$ method is at the worst case we will have an array of size $m$ to store $\sqrt{m}$ elements. This ratio gets worse and worse as $m$ increases. To contrast with the other methods an array of size $m(1+\alpha)$ must have a minimum of $m$ elements meaning this ratio is fixed for all $n$.

(e) As with the other questions we will start by figuring out the size of the last insert then sum the runtimes of all previous inserts. Because this array grows by $\alpha$ every size change, the size of the array will be of the form $1 + b\alpha$ for some $b$ so the size of the final array is $1 + b\alpha$ for the smallest integer $b$ such that

$$(b-1)\alpha + 1 < n \leq b\alpha + 1$$

$$(b-1)\alpha < n - 1 \leq b\alpha$$

$$-1 < \frac{n-1}{\alpha} - b \leq 0$$

So because $b$ is an integer

$$b = \lceil \frac{n-1}{\alpha} \rceil$$

the size of our final array will be

$$\lceil \frac{n-1}{\alpha} \rceil \alpha + 1 = \lceil \frac{n}{\alpha} \rceil \alpha$$

Because the runtime on the last step is the same as the amount of memory allocated on the step before. The total runtime for all realocaiton steps is

$$\sum_{i=0}^{\lceil \frac{n}{\alpha} \rceil - 1} i\alpha$$

which is on the order of $n^2$.

6. First lets prove a smaller lemma, $q$ will always have either the same $x$ coordinate or the same $y$ coordinate as $p$.

*Proof.* Assume $q$ is the road point closest to $p$. Let $q = (q_x, q_y), p = (p_x, p_y)$ Assume for the sake of contradiction that $q_x \neq p_x$ and $p_y \neq q_y$

$$d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

If $q$ is located on a road that runs on the $x$-axis then the point $q' = (p_x, q_y)$ must also be located on that road. We can see that $q'$ is closer to $p$ than $q$

$$d(p,q') = \sqrt{(p_y - q_y)^2} < \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} = d(p,q)$$

If $q$ is located on a road that runs on the $y$-axis then the point $q' = (q_x, p_y)$ is closer to $p$ by the same logic. □

This lemma together with the fact that $q$ is exactly $d$ distance away lets us conclude that there are only 4 possible positions for $q$.

$$q = (p_x + d, p_y) \text{ or } (p_x - d, p_y) \text{ or } (p_x, p_y + d) \text{ or } (p_x, p_y - d)$$

So to find $q$ we simply have to visit all four points, and that can be done by traveling to one of the four points and continuing in a counterclockwise manner to the other three until we hit $q$. This gives us a maximum distance traveled of $(1 + 3\sqrt{2})d \to c = (1 + 3\sqrt{2})$.

7. We know that for our points $q = (q_x, q_y), p = (p_x, p_y)$.

$$d = \sqrt{(q_x - p_x)^2 + (q_y - p_x)^2}$$

$$d^2 = (q_x - p_x)^2 + (q_y - p_x)^2$$

So $q$ is a point in the circle of radius $d$ around $p$. This means if we swim out to a point on that circle and swim around along it we are guaranteed to hit $q$ at some stage. Therefore we will have to swim a maximum distance of $d + 2\pi d = (1 + 2\pi)d \to c = (1 + 2\pi)$.

8. Let's step thorugh our algorithm considering a step to be every time we cross a new possible location for the bus stop.

   - If $d$ is a distance of 1 East $d$ then $w = d = 1$
   - If $d$ is a distance of 1 West we must walk an additional 2 units from our endpoint in the last step for a total $w = 3$.
   - If $d$ is a distance of 2 West we must walk an additional 2 units from our endpoint in the last step for a total $w = 6$.
   - If $d$ is a distance of 2 we need to walk 4 additional steps West $w = 10$.
   - If $d$ is a distance of 3 East $w = 15$.
   - If $d$ is a distance of 3 West $w = 21$.

Note that if $d$ is East of $n$ we will encounter it on step $2d-1$. and that if $d$ is West of $n$ we will encounter it on step $2d$. So let $w_d$ be the $w_s$ value for a particular step $s$ we can see that

$$\begin{aligned} w_s &= w_{s-1} + s \\ &= w_{s-2} + (s-1) + s \\ &= 1 + ... + (s-1) + s \\ &= \sum_{i=1}^{s} i = 1/2(s+1)s. \end{aligned}$$

Therefore if the bus stop is East of us $w = 1/2((2d-1)+1)(2d-1) = d(2d-1)$ and if it's to the West $w = (2d+1)d$. We can see from this that the best case scenario $d/w$ is 1 (when the bus stop in one foot East of us). And $d/w$ gets worse and worse as $d$ increases. Also because $w$ is a polynomial of degree 2 of $d$, $O(w) = \Omega(w) = n^2$.

9. For any $d$ assume $d$ is a distance east, we seek to find the distance walked on the step we passed $d$[4]. Define a step to be to be walkeing from the starting point out East or West then back to the starting point. When you pass the bus stop $d$ distance away it must be the case that $m \geq d$. First group in the two steps before it it we walked a distance of $d/4$ from the starting point out east then a distance of $d/2$ out wast then a distance of $d/4$ back to the starting point we can group these two steps together as contributing $d$ to our total distance week can group then two steps before that together in the same way, we will walk a distance of $d/4$ those two. We contiune in this way until $d = 3$, the distance walked on the first pair of turns. Put in symbols

$$w = d + \frac{d}{4} + \frac{d}{16} + ... + 3 = \sum_{i=1}^{\lceil \frac{\log_4 d}{2} \rceil} (\frac{1}{4})^i d$$

We can see that $O(w(d)) = \Omega(w(d)) = n$

10. To accomplish this I would use a hash function to charcterize files so that way we only have to check files with identical hashes. The hashing algorithm we design accepts a parameter $s$ and is as follows,

    (a) Find the approximate size of the file in bits. Let this number be $n$.
    (b) Read every $m$th byte from the file where $m = n/s$.
    (c) Save these bits to a hash, there should be approximately $s$ bits in each hash.

    The main algorithm is

    (a) Create 16 bit characteristic hashes for every file on $PC_1$ and $PC_2$ and also save the location of the file each hash corresponds to.
    (b) Send every hash from $PC_1$ to $PC_2$, then have $PC_2$ check every hash from $PC_1$ and see if there is a matching hash on on $PC_1$. For $n$ hashes this can be done in $n\log(n)$ time by storing the hashes from $PC_1$ in an ordered binary tree.

[4]For the simplicity we are calculateing the distance you would have traveled had you gone all the way to the end of your range

(c) For every characteristic hash of files on $PC_2$, if we get the same hash of a file on $PC_1$, make a 64 bit hash of the same file on both computers and compare the hashes again. If they are still the same have $PC_1$ send over the full file and compare it to the full file on $PC_2$. If they are identical, report it.

11. The hash function would be as follows. Let $P_i$ be some constant large prime.

    (a) Select $n$ pixels call them $a_1, ..., a_n$ in the following manner $a_m = M[P_1^m \bmod w, P_2^m \bmod h]$ (for $w, h$ with width and height of $M$ respectively).

    (b) Our final hash will be

    $$h(M) = \lfloor P_1 \frac{a_1}{a_2} \rfloor + \lfloor P_2 \frac{a_3}{a_4} \rfloor + ... + \lfloor P_{n/2} \frac{a_{n-1}}{a_n} \rfloor \bmod S$$

    Where $S$ is the desired number of unique hashes.

If $M$ and $M'$ are the same image, $M[i,j] = \beta M'[i,j] \forall i, j$. Let $a_i$ be the $i$th pixel selected from $M$ and let $a_i'$ be the $i$th pixel from $M'$. Because both images have the same width and height, $a_i$ must be from the pixel from the same $i, j$ location as $a_i'$ so $a_i' = a_i \beta$ meaning the hash

$$\begin{aligned} h(M') &= \lfloor P_1 \frac{a_1'}{a_2'} \rfloor + \lfloor P_2 \frac{a_3'}{a_4'} \rfloor + ... + \lfloor P_{n/2} \frac{a_{n-1}'}{a_n'} \rfloor \bmod S \\ &= \lfloor P_1 \frac{a_1 \beta}{a_2 \beta} \rfloor + \lfloor P_2 \frac{a_3 \beta}{a_4 \beta} \rfloor + ... + \lfloor P_{n/2} \frac{a_{n-1} \beta}{a_n \beta} \rfloor \bmod S \\ &= \lfloor P_1 \frac{a_1}{a_2} \rfloor + \lfloor P_2 \frac{a_3}{a_4} \rfloor + ... + \lfloor P_{n/2} \frac{a_{n-1}}{a_n} \rfloor \bmod S \\ &= h(M) \end{aligned}$$

So two identical images will always hash to the same value. We can simply use this hash function along with the method described in the previous question to find identical images.