Homework 4 - Proofs & Counting

Section 6 - Trees & Recursion

Problem 8.4

Prove that the number of binary palindromes of length $2k+1$ is $2^{(k+1)}$ for all $k \geq 0$.

Base Case

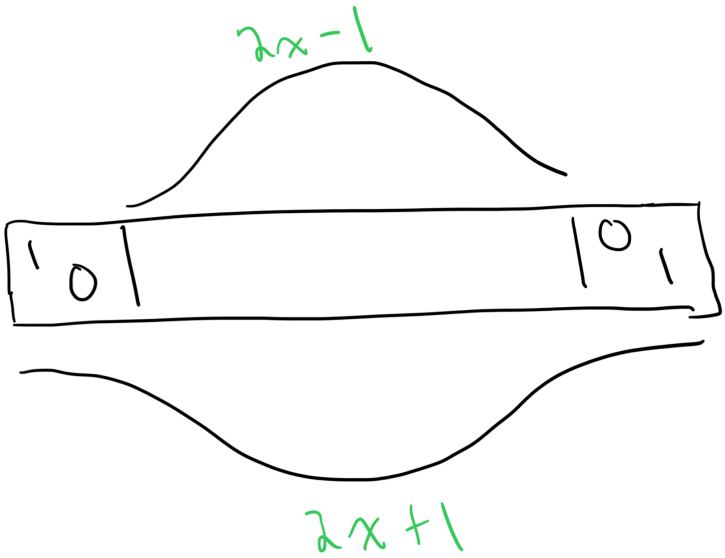
Just by understanding bit logic, we know that the number zero has two possible ways to show it, either 0 or 1. Showing this via the provided formula looks like this: We assume that k is zero, since it is the smallest value that complies with the parameters of this proof. $2(0)+1$ is equal to one bit and $2^{(0)+1}$ is equal to two options. This logically follows because the bit options for one is either 0 or 1. Both of these provide the same answer, which shows that our formula is correct.

Hypothesis

We assume that this property works for any $k < x$ or, in other terms $k = x - 1$.

Inductive Step

To begin the inductive process, we must first replace all k-variables with x so that the new formula looks as so: $2x+1$ and also 2^{x+1}. We are now trying to prove that this formula works for x. We first need to understand that by removing the first and last bit from the sequence from palindromes of length $2x+1$, we get a smaller sequence, formulated by $2^{(x-1)+1}$ because we substituted the hypothesis in for x. This is done in order to account for the equation that has the prepending and appending bits removed. We add a one to the end of the exponent because the palindromes that we are accounting for must have an odd length, and adding a 1 ensures since the central sequence is not getting doubled. The actual, larger, equation with both ends reattached must account for a doubling in size because either one or zero could be added to the ends. Therefore, we multiply $2^{(x-1)+1}$, which can be reduced down to just 2^x, by two, which is $2^x + 2^1$, resulting in 2^{x+1}. This is the equation that we were originally try to prove for.



Problem 8.13

Consider the following algorithm:

```
int halfIt (int n) {
        if (n > 0) {
                return 1 + halfIt(n/2) ;
        } else {
                return 0;
        }
}
```

a.

What does `halfIt(n)` return? Your answer should be a function of n.

N	Return
0	0
1	1
2	2
4	3
8	4
16	5
32	6
N	$\lfloor Log_2 n + 1 \rfloor$

b.

Prove that the algorithm is correct.

Base Case
We assume that $n = 1$ since it is the smallest number that complies with the parameters of the equation. If $n = 1$, then the algorithm is $\lfloor Log_2(1) + 1 \rfloor$ which returns one. This is obvious to see by following the logic in the original program, but the fundamental equation upholds the same conclusion. proves the same. However, this equation does not work for $n \leq 0$, which we talked about and determined that I would only lose a point for not having the algorithm work for such a case.

Hypothesis
We assume that this algorithm works for $n < x$.

Inductive Step
Let n be substituted for x: $\lfloor Log_2 x + 1 \rfloor$. We are now trying to prove that this formula works for x. The recursive case of `halfIt` sees us dividing x by two, which we then substitute for x showing the equation: $\lfloor Log_2 \frac{x}{2} + 1 \rfloor + 1$. We divide by two and add one because that is what the recursive call of `halfIt` does. We then simplify by using the quotient property of logarithms, separating $Log_2 \frac{x}{2}$ into $Log_2(x) - Log_2(2)$, which becomes $\lfloor Log_2(x) - Log_2(2) + 1 \rfloor + 1$. Then, since $Log_2(2) = 1$, we can show $\lfloor Log_2(x) \rfloor + 1$. We can the move the 1 inside of the floored equation, which gives us $\lfloor Log_2(x) + 1 \rfloor$ which is what we were originally trying to prove for.

c.

What is the complexity of halfIt(n)?

This complexity for halfIt(n) would be $Cost(N) = Cost(N/2) + C$ because the n is being halved if the number is greater than zero, which we determined in our hypothesis.

1. $Cost(N/2) + C \rightarrow$ Unwind into $Cost(N/4) + C) + C$, which is then simplified.
2. $Cost(N/4) + 2C \rightarrow$ Unwind into $Cost(N/8) + C) + C$, which is simplified into.
3. $Cost(N/8) + 3C \rightarrow$ This can be continued, unwinding and simplifying, a total of $log_2 N$ times.

Eventually, n reaches zero, meaning that the recurrence relation is $O(LogN)$. The reason that this algorithm is $LogN$ is because we are roughly halving `halfIt` during each recursive call and logarithms fundamentally calculate the number of times a number must be divided until we it reaches 1, which is, in turn, what this algorithm is calculating. This can be further confirmed by comparing the time complexity equation to our recursive equation found above: both end up being close to $LogN$.

Problem 8.15

Assuming the priests can move one disk per second, that they started moving disks 6000 years ago, and that the legend of the Towers of Hanoi is true, when will the world end?

The minimum number of moves can be determined with the formula: $2^N - 1$. In our case, we have sixty-four disks, which means that there are $1.8446744074E19$ minimum moves: $2^{64} - 1 = 1.8446744074E19$. This means that the world will end in $584,868,233,164.2358909321$ years: $1.8446744074E19/3.154e+7 = 584,868,233,164.2358909321$ ($3.154e+7$ is the amount of seconds in a year). Obviously, the 6,000 year head start does not matter at all, and the priests have a long way to go assuming they move each disk in the best case scenario.

Problem 9.4

How many alphabetic strings are there whose length is at most 5?

The wording "at most" is important here, since this indicates that we need to account for strings that are ≤ 5, not just five. Therefore, the answer will be $387,659,011$: $52^5 + 52^4 + 52^3 + 52^2 + 51^1 = 387,659,011$.

Problem 9.11

For what value(s) of k is $(18, k)$ largest? smallest?

Smallest: $(18, 0)$
Largest: $(18, 9)$

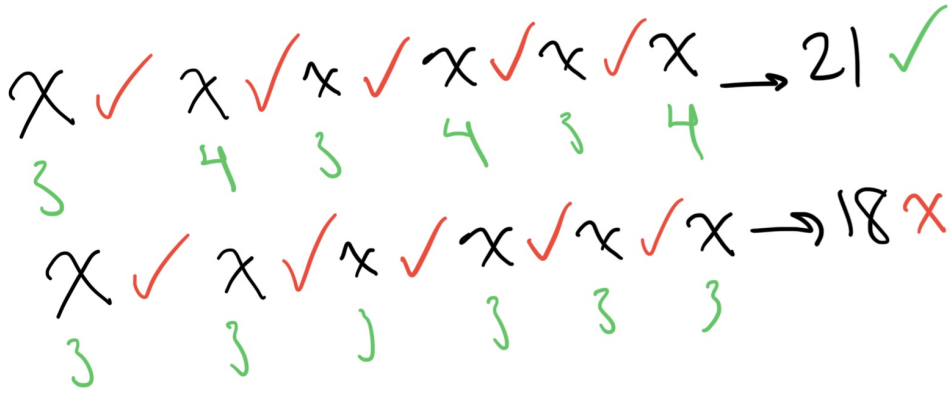
Problem 9.16

Suppose that the letters of the English alphabet are listed in an arbitrary order.

a.

Prove that there must be four consecutive consonants.
If you have to have five vowels, the best way you can can space the list out is by having consonants at the end. By taking twenty-one, the number of consonants, divided by the six different opportunities you would have to place consonants in between five vowels (visually demonstrated by the photo below), you get 3.5, meaning there is space in the series for ≥ 3.5 consonants separated by vowels. Therefore, we would have to have at least one series of $\geq$ four consonants to reach twenty-one.

This photo shows the most effective way of arranging consonants and vowels (vowels displayed with ✔ and consonants displayed with x). If you were to replace each x with a series of three consonants, you would only add up to eighteen, which is less than 21. However, if you are to intersperse series of four consonants, you would be able to add up to 21.



- This shows that you cannot do it with only threes, you must have at least one series of four consonants.

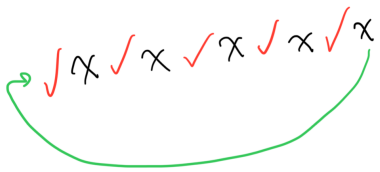
b.

Give a list to show that there need not be five consecutive consonants:
$BCD\mathbf{E}FGHJ\mathbf{I}KLNM\mathbf{O}PQRS\mathbf{U}TVWX\mathbf{A}YZ$

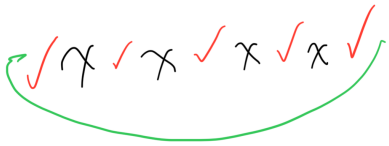
c.

Suppose that all the letters are arranged in a circle. Prove that there must be five consecutive consonants. There are three possible ways for the consonants and vowels to be arranged (vowels displayed with ✔ and consonants displayed with x)

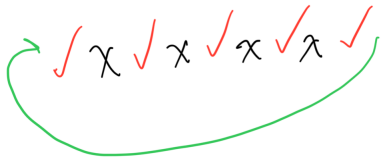
1. Consonant at the beginning and vowel at the end **OR** vowel and the beginning and consonant at the end.



2. Consonant at the beginning and consonant at the end.



3. Consonant at the beginning and end.



Subtracting the total number of letters in the alphabet, 26, by the five vowels, you get 21 which is the total number of consonants. Replacing each of the x's in the above photos with a series of four consonant shows that in, scenario one you would have a total of 20 consonants, in scenario two you would have a total of 16 consonant, in scenario three you would have a total of 20 consonants. This reveals that, no matter which way you arrange it, you would need to have a series of five in order to reach the total of 21 consonants.

Problem 9.25(a-d)

Password cracking is the process of determining someone's password, typically using a computer. One way to crack passwords is to perform an exhaustive search that tries every possible string of a given length until it (hopefully) finds it. Assume your computer can test $10,000,000$ passwords per second. How long would it take to crack passwords with the following restrictions? Give answers in seconds, minutes, hours, days, or years depending on how large the answer is (e.g. $12,344,440$ seconds isn't very helpful). Start by determining how many possible passwords there are in each case.

a.

8 lower-case alphabetic characters: $26^8 = 208,827,064,576$ which equates to $208,827,064,576/10,000,000)/(60)/(60) = 5.8007517938$ hours.

b.

8 alphabetic characters (upper or lower):
$52^8 = 53,459,728,531,456, which equates to $(53,459,728,531,456/10,000,000)/(60)/(60)/(24) = 61.8746858003$

days.

c.

8 alphabetic (upper or lower) and numeric characters: $62^8 = 218,340,105,584,896$, which equates to $(218,340,105,584,896/10,000,000)/(60)/(60)/(24) = 252.7084555381$ days.

d.

8 alphabetic (upper or lower), numeric characters, and special characters (assume there are 32 allowable special characters). $92^8 = 5.1321887314E15$, which equates to $(5.1321887314E15/10,000,000)/(60)/(60)/(24)/(365) = 16.2740637094$ years.

Problem 9.28(b-d)

In March of every year people fill out brackets for the NCAA Basketball Tournament. They pick the winner of each game in each round. We will assume the tournament starts with 64 teams (it has become a little more complicated than this recently). The first round of the tournament consists of 32 games, the second 16 games, the third 8, the fourth 4, the fifth 2, and the final 1. So the total number of games is 32 + 16 + 8 + 4 + 2 + 1 = 63. You can arrive at the number of games in a different way. Every game has a loser who is out of the tournament. Since only 1 of the 64 teams remains at the end, there must be 63 losers, so there must be 63 games. Notice that we can also write 1 + 2 + 4 + 8 + 16 + 32 = 63 as $\sum_{n=1}^{10} n$

b.

When you fill out a bracket you are picking who you think the winner will be of each game. How many different ways are there to fill out a bracket? (Hint: If you think about this in the proper way, this is pretty easy.)
Since there are two options for each game, either winning or losing, and there are sixty three games, the following formula gives us our answer: $2^{63} = 9.2233720369E18$

c.

If everyone on the planet (7,000,000,000) filled out a bracket, is it guaranteed that two people will have the same bracket? Explain.
No, because there are around nine-**quintillion** ways to fill out a bracket and there are only seven-billion people on the planet. $9.2233720369E18 > 7,000,000,000.$

d.

Assume that everyone on the planet fills out k different brackets and that no brackets are repeated (either by an individual or by anybody else). How large would k have to be before it is guaranteed that somebody has a bracket that correctly predicts the winner of every game?
If you were to take the number of possible brackets and divide it by the amount of people who are currently alive, there would be $1,317,624,576.7$ possible brackets for each person:
$9.2233720369E18/7,000,000,000 = 1,317,624,576.7$