B06902102, 羅費南

Non Programming Part

Problem 1

1. Time complexity of O(n) is reached, since I only used two for loops to solve the problem, they are not related to each other, and the if's inside only use O(1) complexity time. The space complexity is reached, since no other variable other than the ones provided are used, and is independent of the input size.

```
Algorithm Sort_Pusheen_1 ( Pusheen_Array )
int n0 = 0, n1 = 0, n2 = 0;
for (i=0; i< length ( Pusheen_Array ); i ++){
    if (Pusheen_Array[i]=="0")
        n0++;
    if (Pusheen_Array[i]=="1")
        n1++;
}
for (i=0; i< length ( Pusheen_Array ); i ++){
    if (i<n0)
        Pusheen_Array[i]="0";
    else
        if (i>=n0 && i<n0+n1)
            Pusheen_Array[i]="1";
        else
            Pusheen_Array[i]="2";
}
return Pusheen_Array;
```

2. It meets the constraints since I didn't declare any more variables, so the size is O(1), and is still O(n) time because each loop runs in O(n) time, and added up is still O(n).

```
Algorithm Sort_Pusheen_2 ( Pusheen_Array )
int n0 = 0, n2 = length ( Pusheen_Array ) -1;
for (i=0; i< length ( Pusheen_Array ); i ++){
    while (Pusheen_Array[n0]==0)
        n0++;
    while (Pusheen_Array[n2]==2)
        n2--;
    if (Pusheen_Array[i]==0)
        Swap(i,n0);
    if (Pusheen_Array[i]==2)
        Swap(i,n2);
}
return Pusheen_Array;
```

3. The number of variables is independent of the size of the input, thus size O(1), and because it only utilizes one loop that traverses the entire array of data, the time complexity is only O(n). As we already know one of the values when we check each one of them, we can use a swap without using an extra variable for it. To avoid counting extra items for the swap, we denote n4 as being a flag variable, so as if the loop has already swapped one variable, it doesn't count an extra number to sort.

```
Algorithm Sort_Pusheen_3 ( Pusheen_Array )
int n0 = -1, n1 = -1, n2 = -1, n3 = -1, n4 = -1;
for (i=0; i< length ( Pusheen_Array ); i ++){
    n4=-1;
    if (Pusheen_Array[i]==0) {
        n0++,n4++;
        Pusheen_Array[i]=Pusheen_Array[n0];
        Pusheen_Array[n0]=0;
    }
    if (Pusheen_Array[i]==1) {
        if (n4==-1)
            n1++,n4++;
        Pusheen_Array[i]=Pusheen_Array[n0+n1+1];
        Pusheen_Array[n0+n1+1]=1;
    }
    if (Pusheen_Array[i]==2) {
        if (n4==-1)
        n2++,n4++;
        Pusheen_Array[i]=Pusheen_Array[n0+n1+n2+2];
        Pusheen_Array[n0+n1+n2+2]=2;
    }
    if (Pusheen_Array[i]==3) {
        if (n4==-1)
            n3++,n4++;
        Pusheen_Array[i]=Pusheen_Array[n0+n1+n2+n3+3];
        Pusheen_Array[n0+n1+n2+n3+3]=3;
    }
}
return Pusheen_Array;
```

Problem 2

1. To modify the merge sort so it can work for the rabbits, we need to ask each rabbit that is not the rabbits we are comparing which one is bigger. Because it is guaranteed that over half of the rabbits are good rabbits, if we count the results of who says bigger or who smaller, the one who has the majority is guaranteed to be the right answer. Then we swap accordingly. We do that for every swap we need to do. Because we will do N questions per pass, and we do N passes, the complexity for questions is N^2. The rest of the complexity for merge is lg(n), so the total complexity is indeed $O(n^2 lg(n))$.

2. Worst case complexity can be described as follows: suppose we grab the smallest rabbit for each k we grab. For the insertion sort, we ask them O(N) times for their sizes to sort them. Then for the actual sort, is dependent on the size of the k rabbits, so is O(k^2) for the insertion sort. Then, the quicksort is dependent on the entire input, which gives us O(N^2) worst case. The worst case is that we check all the rabbits that are outside the insertion sort, which is N/k times for checking. So, that gives us N/k*O(Nk^2+N^2) worst time, which we can also write as O(N^2+N^3/k).

3. The purpose of line 2 is to swap the elements when rabbit[start] is older than the rabbit[end]. If we remove this line, there cannot be any swapping done in the entire sort, which means nothing moves; thus, you cannot sort anything.

4. The algorithm calls recursively to itself three times; it splits itself into the first two thirds of it, then it takes the last two thirds elements and then sorts the first two thirds elements again. Because it does subsequent 2/3 calls of itself for each call, and each call is the same independent of the total size, we can express the recurrence as follows (note that we can also solve the given equation once we find the solution for the recurrence). https://www.cs.cmu.edu/~eugene/teach/algs00a/works/s3.pdf

$$T(n) = 1 + 3T(\frac{2}{3}n)$$
$$T(n) = 1 + 3 + 9T(\frac{4}{9}n)$$
$$T(n) = 1 + 3 + 9 + 27T(\frac{8}{27}n)$$
$$T(n) = \ldots$$
$$T(n) = 1 + 3 + 3^2 + \ldots + 3^{\log_{\frac{3}{2}} n}$$
$$T(n) = \frac{3^{\log_{\frac{3}{2}} n+1} - 1}{3-1}$$
$$T(n) = \Theta(3^{\log_{\frac{3}{2}} n})$$
$$T(n) = \Theta(3^{\frac{\log_3 n}{\log_3 \frac{3}{2}}})$$
$$T(n) = \Theta(n^{\frac{1}{\log_3 \frac{3}{2}}})$$
$$T(n) = \Theta(n^{2.71})$$

$$\log_{1.5} c = 2.71$$
$$1.5^{2.71} = c$$
$$c = 3.0005$$

Problem 3

1. My student ID is B06902102. The hashing for that string with base 36 and modular 77 is 59. If we have a String “B06902127”, it will make the spurious hit happened, and it doesn’t contain my ID, because the hashing for the substring B06902127 is also 59, but it is not equal to my ID. I came across it by adding another 77 to the value of the hashing of my ID without mod (which is 2*36^1+5*36^0), thus making the “mod 77” of that number also 59.
2.
```
int is_repeated_string (char *X, int n) {
    int *pi = prefix_function (X);
    if (pi[n-1]>=n-pi[n-1] && pi[n-1]%(n-pi[n-1])==0 && n>1)
        return 1; //True
    return 0; //False
}
```

To determine the correctness of the algorithm, we need to examine how the prefix function works. The prefix function gives us the length of the biggest suffix of a string which is also a prefix of said string. That means, for each substring that is found on the string, we find the longest prefix and suffix up to that index. Now, for the entire string, if we want to determine if it follows the pattern of being $(S)^k$, that

means, if we can repeat the S substring k times to get a bigger string X. Now, a suffix cannot be of the same length as the entire string (same goes for the prefix), so for any S string in X we would like to find that satisfies the condition, the length of the biggest prefix/suffix would be the length of the entire string X minus the length of the string S. To determine the length of the string S we can subtract the length of X minus the length of the biggest prefix/suffix, which we get in the n-1 position of pi (assuming the S string satisfies S^k=X). To verify that it's a string S^k, we can mod the subtraction with the n-1 position of pi. If it's 0, then it is perfectly divisible with the length of our presumed S, which means that the string must be of the form S^k.

3.

```
void palindrome (char* X, char* Y, int n) {
    int j=strlen(X)-1;
    int mark=j;
    int found=0;
    for (int i=0;i<j;i++) {
        char A=X[i];
        char B=X[j];
        if (A==B) {
            found=1;
            j--;
            if (mark>i)
                mark=i;
        }
        else {
            if (found==1) {
                found=0;
                i--;
            }
            j=strlen(X)-1;
            mark=j;
        }
    }
    int k=0;
    for (int i = mark-1; i >=0; i--)
        Y[k++]=X[i];
    Y[k]=0;
}
```

The algorithm is correct because it checks for the longest palindrome which is inside of the current string. This longest palindrome has to end at the final position of the checked string, so that when you add characters after the string, the palindrome can be built from there. Mark variable saves the position where the longest palindrome begins, and in the other loop, it saves every character

backwards from mark position to the beginning; but reversed. This is to make it that it appears reversed and as an actual palindrome of the new string.

It meets the time constraint since the first loop runs at worst in O(n) time, and the second loop also runs at worst O(n) time; since they are not nested, the algorithm runs at O(n) time.

https://stackoverflow.com/questions/18732020/add-the-least-amount-of-characters-to-make-a-palindrome

4.

```
bool match(char* P, char* S)
{
    int n=strlen(P);
    int m=strlen(S);
    char Q[n];
    while(i<m) {
        int k=i,j=0,l=0,flag=0;
        if (*P[i]=='*') {
            i++;
            continue;
        }
        while (*P[i]!='*' && i<m) {
            Q[j++]=P[i++];
        }
        if (*P[i]=='*') {
            i++;
            if (i==m)
                return true;
        }
        int* pi =prefix_function(Q);
        while (k<m) {
            if (P[l]==S[k]) {
                l++,k++;
            }
            if (l==n) {
                flag=1;
                break;
            }
            else
                if (k<m && P[l]!=S[k]) {
                    if (l!=0)
                        l=pi[l-1];
                    else
                        k++;
                }
        }
        if (flag==0)
            return false;
    }
    return true;
}
```

The algorithm splits the pattern string each time it finds a * into a new substring. It performs KMP of that substring into the original string until it finds a match. If it reaches the end of the string without finding a match, it returns false. It repeats this process, moving the variable i after the last * found. If * is the last character of the string, it returns true. It runs on O(n+m) time because it does m [O(m)] times KMP algorithm, each time independent from the other. Since KMP is O(n), the entire algorithm runs at O(n+m).

https://stackoverflow.com/questions/39817178/handling-wildcard-operator-in-string-matching-using-kmp-algorithm?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa