A1.2) The moves required for 8 disks are 45 and they are:

T1->T3

T1->T4

T3->T4

T1->T2

T1->T3

T2->T3

T4->T2

T4->T3

T2->T3

T1->T4

T1->T2

T4->T2

T3->T2

T3->T1

T2->T1

T3->T4

T3->T2

T4->T2

T1->T4

T1->T2

T4->T2

T1->T3

T1->T4

T3->T4

T2->T1

T2->T3

T1->T3

T2->T4

T2->T1

T4->T1

T3->T4

T3->T1

T4->T1

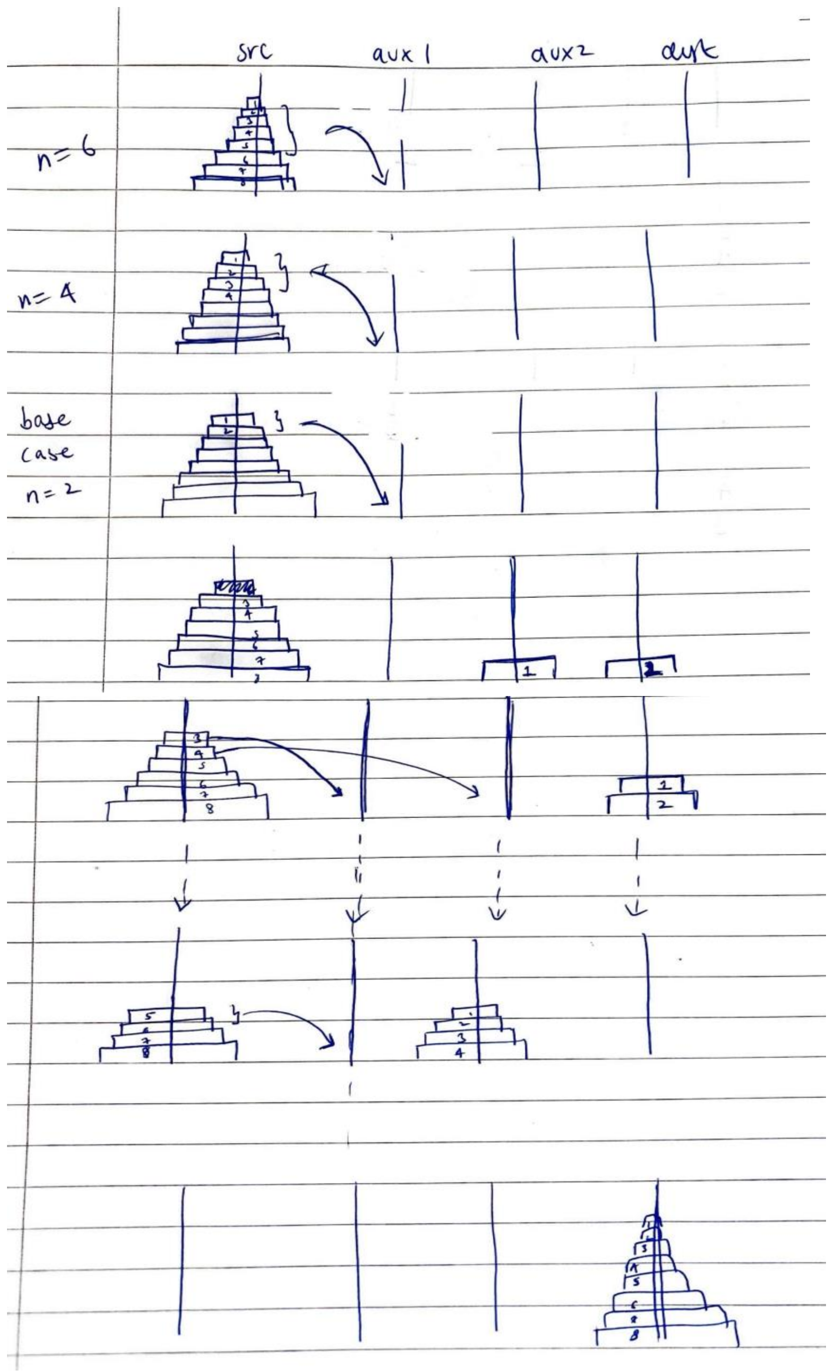
T2->T3

T2->T4

T3->T4

T1->T4
T1->T2
T4->T2
T1->T3
T1->T4
T3->T4
T2->T3
T2->T4
T3->T4

- The function is called recursively for n=6,n=4 to move disk from source to auxiliary 1 until it reaches n=2.
- Once the base case is reached it moves two disks from source to destination using the auxiliary 2 rod.
- It then returns null; to n=4 case
- Moves the discs from source to destination.
- Function is called again for n=4 with source as auxiliary 1 to destination. This is also called recursively until n=2 case.
- It then gets back to n=6 case and last two steps are repeated.



A1.3) Since the traditional tower of Hanoi has 3 pegs the code for the following is –

```
void towerOfHanoi3(int n, string src, string aux, string dest){
    if(n==1){
        cout<<src<<"->"<<dest<<endl;
        return;
    }
    towerOfHanoi3(n-1,src,dest,aux);
    cout<<src<<"->"<<dest<<endl;
    towerOfHanoi3(n-1,aux,src,dest);
}
```

The time complexity can be calculated in this manner:

\# for 3 pegs, $T(n) = \underbrace{2T(n-1)}_{\text{2 recursive calls for (n-1)}} + \underbrace{1}_{\text{1 check}}$

$T(n-1) = 2T(n-2) + 1$

substituting

$T(n) = 4T(n-2) + 3$

$T(n-2) = 2T(n-3) + 1$

substituting

$T(n) = 8T(n-3) + 7$

in general

$T(n) = 2^k T(n-k) + 2^k - 1$

for $n-k=1$: $T(1) = 1$

$T(n) = 2^{n-1} T(1) + 2^{n-1} - 1$

$= 2^{n-1} + 2^{n-1} - 1$

$= 2^n - 1$

$T(n) = O(2^n)$

The code for 4 pegs is:

```
void towerOfHanoi(int n, string src, string aux1, string aux2, string dest){
    if(n<0){
        cout<<"Invalid number of discs"<<endl;
        return;
    }

    if(n==1){
        cout<<src<<"->"<<dest<<endl;
        return;
    }
    if (n == 2) {
        cout<<src<<"->"<<aux2<< endl;
        cout<<src <<"->" <<dest<< endl;
        cout<<aux2<<"->"<<dest<< endl;
        return;

    }

    towerOfHanoi(n-2,src,aux2,dest,aux1);
    cout<<src<<"->"<<aux2<<endl;
    cout<<src<<"->"<<dest<<endl;
    cout<<aux2<<"->"<<dest<<endl;
    towerOfHanoi(n-2,aux1,src,aux2,dest);
}
```

The time complexity for 4 pegs is:

$\#$ for 4 pegs $T(n) = 2T(n-2) + c$

$T(n-2) = 2T(n-4) + c \Rightarrow$ substituting

$T(n) = 2(2T(n-4) + c) + c$

$T(n) = 4T(n-4) + 3c$

$T(n-4) = 2T(n-6) + c \Rightarrow$ substituting

$T(n) = 8T(n-6) + 7c$

$\vdots$

$T(n) = 2^k T(n-2k) + (2^k - 1)c$

$n - 2k = 1 \qquad \frac{n-1}{2} = k$

$T(n) = 2^{\frac{n-1}{2}} T(1) + (2^{\frac{n-1}{2}} - 1)c$

$= \frac{2^{n/2}}{2} + \left(\frac{2^{n/2}}{2} - 1\right)c$

$\therefore T(n) = O(2^{n/2})$

Q2.2) The code for iterative merge sort is:

```
#include <iostream>

using namespace std;

void printArray(int arr[],int size){
    for(int i=0; i<size; i++){
        cout<<arr[i]<<' ';
    }
    cout << endl;
}
```

```
void merge(int arr[],int s,int mid,int e){

    int len1 = (mid + 1) - s;
    int len2 = e - mid;

    int first[len1];
    int second[len2];

    // copying first part of array to array named first
    int k=s;
    for(int i=0;i<len1;i++){
        first[i] = arr[k++];
    }
    // copying second part of array to array named second
    k = mid+1;
    for(int i=0;i<len2;i++){
        second[i] = arr[k++];
    }

    // merge 2 sorted arrays
    int i1=0;
    int i2=0;
    k=s;
    // comparting then adding elements to arr
    while(i1<len1 && i2<len2){
        if(first[i1]>second[i2]){
            arr[k++] = second[i2++];

        }
        else{
            arr[k++] = first[i1++];

        }
    }
    // if any elements left
    while(i1<len1){
        arr[k++] = first[i1++];
    }
    while(i2<len2){
        arr[k++] = second[i2++];
    }
}

int main(){
    int arr[5] = {6,1,23,2,7};
    int size = sizeof(arr)/sizeof(arr[0]);
    int i;
    for(i=2;i<=size;i=i*2){
```

```
        for(int j=0; j<size;j=j+i){
            int e = i+j-1;
            if(e>=size){

                e=size-1;
            }

            merge(arr,j,(j+e)/2,e);
        }
    }
    // for arrays with length not in multiples of 4, last 2 or last element
will be left unsorted
    if(i/2!=size){
        merge(arr,0,i/2-1,size-1);
    }
    printArray(arr,size);
    return 0;
}
```

Taking an example of [4,3,2,1,0]
The first loop divides the list into sub array each of length 2 except the last element (because the length of the array is odd).
[4,3] [2,1] [0]
For each sub array the merge function is called which finds the middle element and merges the sorted arrays from s to mid+1 and mid+1 to e. This step results in array = [3,4,1,2,0].
The second loop divides the array into sub array of length 4.
[3,4,1,2] [0]
Again, the merge function is called, and the array becomes [1,2,3,4,0].
Since i becomes 8 the loop ends.
The last if condition checks if i/2 is not equal to the size. Since its not equal, the merge function is called again with mid-3.
The final array is [0,1,2,3,4].

Q3)

$f_{24} = 2^{2^n}$

$f_{23} = n^n$

$f_{22} = n!$

$f_{21} = 2^{2n}$

$f_{20} = 4^n$

$f_{19} = 3^n$

$f_{18} = n2^n$

$f_{17} = 2^{n+1}$

$f_{16} = 2^n$

$f_{15} = 2^{-1\sqrt{n}}$

$f_{14} = n^{65}$

$f_{13} = n^{64}$

$f_{12} = \binom{n}{64}$

$f_{11} = n\log n$

$f_{10} = \log n!$

$f_9 = n2^{100}$

$f_8 = 3n$

$f_7 = 2n$

$f_6 = \log_2 n$

$f_5 = \log_{10} n$

$f_4 = 2^{2^{100}}$

$f_3 = \frac{1}{\sqrt{n}}\ [n^{-1/2}]$

$f_2 = (\log n)/n$

$f_1 = n^{-1}$