MIDTERM

P1.

Algorithm:

1. Here, our function will take array and it's size as input.
2. Initialize max difference as arr[1]-arr[0]
3. Initialize min_element as arr[0]
4. Now, run a for loop over n elements of array.
5. Check if arr[i]-min_element > maxDiff, if yes then update maxDiff.
6. Check if current element of i is smaller than min_element, if yes then update min_element.
7. Time complexity will be O(n).

Pseudocode:

```
int maxDifference(int arr[], int arr_size)
{
    int maxDiff = arr[1] - arr[0];
    int min_element = arr[0];
    for(int i = 1; i < arr_size; i++)
    {
      if (arr[i] - min_element > maxDiff)
      {
         maxDiff = arr[i] - min_element;
      }
      if (arr[i] < min_element)
      {
         min_element = arr[i];
      }
    }
    return maxDiff;
}
```

Time Complexity : **O(n)**

P3.

Algorithm:

1. Start with the input numbers and then depending on number of digits there are in a set we start fixing one number and swap the rest.
2. If the set with numbers in {1,2,3}, we create 3 sets and fix each of the three numbers in the first index of each set.
3. With the remaining digits in the set, fixing the number in the second index.
4. We keep doing this recursively till we reach a point where n-1 number are fixed and the point the last number will be automatically fixed and them print out.
5. Here, base case is, if 1==r meaning if we reach at the last element then out swapping ends, and we print the value if it was not printed before.
6. To avoid, repetition we can store the value that we are printing and check every time if the value is already printed or not.

Pseudocode:

Input parameters for function are as below

1. string of integers, starting index of string, ending index of string.

```
permute (int string, int 1, int r)
{
    If (1==r) // check if we have any numbers to permute, if not then print
    Store current permutation in array as string; // to check we need helper function
    if(a is not in stored array)
    {
        cout << a << endl;
    }
    else
    {
        For ( int i = 1; i <= r; i++)
        {
            Swap ( a[1], a[i]);     // swap
            Permute (a, 1+1, r)   // Recursive call
            Swap (a[1], a[i]);   //backtrack
```

```
            }
        }
}
```

P4.

Algorithm:

1. Here, we can simply use stack to keep track of parenthesis using stack data structure.
2. Our base case for this algorithm will be if we reach end of string(input) and stack is not empty then we return false, else return true.
3. Another base case will be if string length (input length) is 0 then we return true.
4. For, n>1 we read input and push to stack.
5. When we encounter similar closing parenthesis which is a pair of s.top() then we pop() s.top(). For example if we encounter ‘]’ and s,top()==’[‘ then we pop() s.top().
6. Here, input parameters are string and starting index.

Pseudocode:

```
Void validParenthesis(string s, int n=0)
{
    Stack<char> s;
   If(s.size()==0)
    Return true;
    If (s.empty() && n==s.size())
    {
            Return true;
    }
    Else
    {
            Return false;
    }
    If(s[n] is ‘[‘ or ‘(‘ )
            s.push(s[n])
    else if( s[n] I ‘]’ or ‘)’)
            check if s.top()==’[‘ && s[n]==’]’
```

```
                    s.pop()
                check if s.top()==’(‘ && s[n]==’)’
                    s.pop()
    validParenthesis(s, n+1)
}
```