Problem - A

Efficiency consideration in terms of computation time for Bubble sort and Merge Sort.

Machine Configuration

Processor	:	Intel® Core™ i5-7200U CPU @ 2.50 GHz 2.71 GHz
Installed memory (RAM)	:	8.00 GB(7.89 GB usable)
System type	:	64-bit Operating System, x64-based processor

Solution in C++

Generation of Random Numbers

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;

    cout << "How many numbers do you generate? : ";
    cin >> n;

    freopen("G:\Random Numbers.txt", "w", stdout);

    srand(time(0));

    for(int i = 0; i < n; i++){
        int temp = (rand() % 10001) * 10000;
        temp += 10000+(rand() % 1000);
        if(i & 1) temp *= -1;
        cout << temp << endl;
    }
}
```

Sorting Numbers in Bubble Sort Approach

```
#include <bits/stdc++.h>
using namespace std;

void bubble_sort(int* num, int n){
    bool isSwapped;

    for(int i = 0; i < n-1; i++){
        isSwapped = false;
        for(int j = 0; j < n-1-i; j++){
            if(num[j] > num[j+1]){
                int temp = num[j];
                num[j] = num[j+1];
                num[j+1] = temp;

                isSwapped = true;
            }
        }

        if(!isSwapped) break;
    }
}

int main(){
    int num[50005];
    int n;

    cout << "How many numbers do you want to process?: ";
    cin >> n;

    freopen("G:\Random Numbers.txt", "r", stdin);
    freopen("G:\Output for Bubble Sort.txt", "w", stdout);

    for(int i = 0; i < n; i++){
        scanf("%d", &num[i]);
    }

    clock_t start = clock();

    bubble_sort(num, n);

    clock_t stop = clock();

    double duration = (double) (stop - start);
    duration /= (CLOCKS_PER_SEC*1.00);

    for(int i = 0; i < n; i++){
        printf("%d\n", num[i]);
    }

    cout << "Time required for bubble sort for " << n << " data: "
        << duration << " seconds." << endl;
}
```

Sorting Numbers in Merge Sort Approach

```
#include <bits/stdc++.h>
using namespace std;

void merge_sort(int* num, int lo, int hi){
    if(lo == hi) return;
    int mid = lo + (hi - lo) / 2;

    merge_sort(num, lo, mid);
    merge_sort(num, mid+1, hi);

    int i, j, k, kk;

    int temp[hi-lo+9];

    for(i = lo, j = mid+1, k = 0, kk = lo; kk <= hi; k++, kk++){
        if(i == mid+1) temp[k] = num[j++];
        else if(j == hi+1) temp[k] = num[i++];
        else if(num[i] < num[j]) temp[k] = num[i++];
        else temp[k] = num[j++];
    }

    for(int xx = lo, yy = 0; xx <= hi; xx++, yy++) num[xx] =
temp[yy];
}

int main(){
    int num[50005];
    int n;

    cout << "How many numbers do you want to process?: ";
    cin >> n;

    freopen("G:\Random Numbers.txt", "r", stdin);
    freopen("G:\Output for Merge Sort.txt", "w", stdout);

    for(int i = 0; i < n; i++){
        scanf("%d", &num[i]);
    }

    clock_t start = clock();

    merge_sort(num, 0, n-1);

    clock_t stop = clock();
    double duration = (double) (stop - start);
    duration /= (CLOCKS_PER_SEC*1.00);

    for(int i = 0; i < n; i++){
        printf("%d\n", num[i]);
    }

    cout << "Time required for merge sort for " << n << " data: "
        << duration << " seconds." << endl;
}
```

Theoretical Complexity for Bubble Sort

$n + (n - 1) + (n - 2) + \ldots + 1$

$= (n*(n + 1))/2$

$= O(n^2)$

Theoretical Complexity of Merge Sort

```
merge_sort(array, lo, hi){
    mid = lo + (hi - lo) / 2;                        c
    merge_sort(array, lo, mid);                      T(n / 2)
    merge_sort(array, mid+1, hi);                    T(n / 2)
    //looping lo to hi for merging two arrays        n
}
```

$T(n) = 2T(n/2) + cn$

$= 2\{2T(n/4) + c.(n/2)\} + cn$

$= 4T(n/4) + 2cn$

$= 2^k T(n/2^k) + kcn$

Let, $k = \log_2 n$

$= nT(1) + cn\log_2 n$

$= n + cn\log_2 n$

Hence, overall complexity: $O(n\log_2 n)$

Experimental Result

N	Bubble Sort	Merge Sort
5000	0.062 seconds	0.001 seconds
10000	0.202 seconds	0.001 seconds
25000	1.437 seconds	0.001 seconds

N	Bubble Sort	Merge Sort
40000	3.615 seconds	0.012 seconds
50000	5. 695 seconds	0.017 seconds

Comment

As for every number of data we encountered, merge sort took significantly less time than bubble sort, we can conclude that merge sort is more time efficient than bubble sort.

Problem - B

Finding a better approach than the brute force technique to find the sum of three numbers that adds up to zero.

Machine Configuration

Processor	:	Intel® Core™ i5-7200U CPU @ 2.50 GHz 2.71 GHz
Installed memory (RAM)	:	8.00 GB(7.89 GB usable)
System type	:	64-bit Operating System, x64-based processor

Solution in C++

Generation of Random Numbers

Random numbers were generated with the same solution of random number generation in Problem - A.

Comparison of Brute Force and Optimized Approach

```
#include <bits/stdc++.h>
using namespace std;

void merge_sort(int* num, int lo, int hi){
    if(lo == hi) return;
    int mid = lo + (hi - lo) / 2;

    merge_sort(num, lo, mid);
    merge_sort(num, mid+1, hi);
    int i, j, k, kk;

    int temp[hi-lo+9];

    for(i = lo, j = mid+1, k = 0, kk = lo; kk <= hi; k++, kk++){
        if(i == mid+1) temp[k] = num[j++];
        else if(j == hi+1) temp[k] = num[i++];
        else if(num[i] < num[j]) temp[k] = num[i++];
        else temp[k] = num[j++];
    }

    for(int xx = lo, yy = 0; xx <= hi; xx++, yy++) num[xx] =
temp[yy];
}

int bin_search(int* num, int lo, int hi, int target){
    if(lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if(num[mid] == target) return mid;
    else if(num[mid] < target) return bin_search(num, mid+1, hi,
target);
    else return bin_search(num, lo, mid-1, target);
}

int main(){
    int num[100005];

    int index;

    cout << "How many numbers do you want to process?: ";
    cin >> index;

    freopen("G:\Random Numbers.txt", "r", stdin);
    freopen("G:\Output for Algorithm Comparison.txt", "w", stdout);

    for(int i = 0; i < index; i++)
        scanf("%d", &num[i]);

    cout << "Bruteforce approach:\n";

    clock_t start1 = clock();

    for(int i = 0; i < index-2; i++){
        for(int j = i+1; j < index-1; j++){
            for(int k = j+1; k < index; k++){
                if(num[i] + num[j] + num[k] == 0){
                    cout << num[i] << " " << num[j] << " " << num[k]
<< endl;
                }
            }
        }
    }

    clock_t stop1 = clock();

    double duration1 = (double)(stop1 - start1);
    duration1 /= CLOCKS_PER_SEC;

    merge_sort(num, 0, index-1);

    cout << "Optimised approach:\n";

    clock_t start2 = clock();

    for(int i = 0; i < index-2; i++){
        for(int j = i+1; j < index-1; j++){
            int tempsum = num[i] + num[j];
            tempsum *= -1;
            int t2 = bin_search(num, j+1, index-1, tempsum);
            if(t2 != -1){
                cout << num[i] << " " << num[j] << " " << num[t2] <<
endl;
            }
        }
    }
    clock_t stop2 = clock();

    double duration2 = double(stop2 - start2);
    duration2 /= CLOCKS_PER_SEC;

    cout << "Total time consumed for bruteforce approach: ";
    printf("%.19lf seconds.\n", duration1);
    cout << "Total time consumed for optimized approach: ";
    printf("%.19lf seconds.\n", duration2);
}
```

Theoretical Complexity

Brute Force Approach:
(n-3)*(n-2)*(n-1)*c
= O(n^3)
Optimised Approach:
(n-3)*(n-2)*log n
= O (n^2 log n)

Experimental Result

N	Brute Force Approach	Optimized Approach
200	0.0010	0.0000
300	0.0129	0.0000
400	0.0299	0.0000
500	0.0449	0.0170
600	0.1860	0.0179
900	0.3400	0.0179
1000	0.7419	0.0660
1500	2.6360	0.1650
2000	5.7839	0.2000
5000	98.0100	2.3913

Comment

As we see, the difference between the consumed time between brute force approach and optimized approach is increasing with the increase of the number of data. Hence, we can conclude that, brute force approach will be bad algorithm for the process for big number of data in comparison with the optimized approach.