1.

k路归并与二路归并的算法过程类似，主要是在k路归并的处理上。若是用朴素的遍历，每次从k个数组中寻求最小值，共有n个数，这时的复杂度是 nk， 若是采用最小堆的方式，可以优化这一过程。由于堆的大小不超过k, 则n个数的复杂度为 $nlog_2k$。

当进行二路归并时，共需要归并 $ceil(log_2n)$ 层；类似地，当进行k路归并时，共需要归并 $ceil(log_kn)$ 层，由上可得每一层复杂度为 $nlog_2k$, 则复杂度为$O(log_kn*nlog_2k)=O(nlog_2n)$ 。

当归并到范围小于k时，由于数字较少，采用了选择排序来处理。具体的实现过程可以见下面代码(最小堆为手写所以可能略显复杂):

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e7 + 20;
//num存从文件里读出的数据，每次取不同k时，需要将num的数据拷贝到a数组
//由于归并的过程中，需要用到两个数组，所以a数组和b数组循环使用
//k个数组归并时，Left，Right双指针来表示某个数组还未入堆的范围
//h是最小堆，记录的是最小的数，h2记录的是h对应的数来自哪个数组
int num[maxn], a[maxn], b[maxn], Left[200], Right[200], h[200], h2[200], n;

void siftdown(int i,int n)                          //对于存在[0, n-1]的最小堆，将第i位调整
{
    int j, t = h[i], t2 = h2[i];
    while((j = 2 * i + 1) < n)
    {
        if(j < n - 1 && h[j] > h[j+1]) j++;
        if(t > h[j])
        {
            h[i] = h[j];  h2[i] = h2[j];
            i = j;
        }
        else break;
    }
    h[i]= t;  h2[i] = t2;
}

void k_sort(int l, int r, int k, int *a, int *b)      //上一步归并的结果在b数组，这一步归并记录到a数组
{
    if(r - l <= k)                                    //当数字较少时(小于等于K),进行选择排序
    {
        for(int i = l; i < r; i++) a[i] = num[i];
        for(int i = l; i < r - 1; i++)
        {
            int mx = i;
            for(int j = i + 1; j < r; j++)
                if(a[j] < a[mx]) mx = j;
            swap(a[i], a[mx]);
        }
        return;
    }
```

```
    int gap = (r - l) / k;
    for(int i = 0; i < k - 1; i++)        //将[L, r)范围的数分成k组，先分别归并，然后合并
    {
        k_sort(l + i * gap, l + (i + 1) * gap, k, b, a);
    }
    k_sort(l + (k - 1) * gap, r, k, b, a);

    for(int i = 0; i < k; i++)                    //确定k个数组的起始与结尾，同时将每个数组的第一个入栈
    {
        Left[i] = l + i * gap;
        if(i < k - 1) Right[i] = Left[i] + gap;
        else Right[i] = r;
        h[i] = b[Left[i]];
        h2[i] = i;
    }

    for(int i = (k - 2) / 2; i >= 0; i--)              //将已经有k个数的堆调整成最小堆
        siftdown(i, k);

    int cur = k;
    for(int i = l; i < r - 1; i++)
    {
        a[i] = h[0];                                   //每次取堆顶，然后将对应数组下一个数入堆
        if(++Left[h2[0]] < Right[h2[0]])
            h[0] = b[Left[h2[0]]];
        else                                           //若这个数组取完了
        {
            h[0] = h[cur-1];
            h2[0] = h2[cur-1];
            cur--;
        }
        siftdown(0, cur);
    }
    a[r-1] = h[0];
}

int main()
{
    FILE *file = fopen("data1m.txt","r");                  //先从文件中读取数据到num数组
    fscanf(file, "%d", &n);
    for(int i = 0; i < n; i++)
        fscanf(file, "%d", &num[i]);
    fclose(file);

    for(int k = 2; k <= 60; k++)                           //对K进行枚举
    {
        for(int i = 0; i < n; i++)
            a[i] = num[i];
        clock_t start = clock();
        k_sort(0, n, k, a, b);
        clock_t finish = clock();
        printf("k = %d, 执行时间为: %d\n", k, finish - start);
        //for(int i = 0; i < n; i++) printf("%d ", a[i]);
```

```
        putchar('\n');
    }
}
```

1e6的数据，最后的结果为：

k	Time （ms)
2	292
3	200
4	180
5	170
6	170
7	170
8	160
9	167
10	170
11	161
12	160
13	170
14	180
15	180
20	160
25	170
30	171
35	190
40	182
50	161
60	174
70	202
80	206
90	180
100	250
125	223
150	200
175	200
200	190

发现当k = 8时取得最小值，而其实所测试的k值的时间差别并不是很大，1e6数量级时大部分都在160-200ms波动。分析可能原因的是根据上面的计算复杂度其实是相似的，当k比较大时会有更多的进行选择排序，当k比较小的时候又会有更多的递归开销，所以相差不大。

2.

首先第一种方法是容斥的方法，即去重。首先进行二重遍历，将每个二元组的和都可以遍历到，用map记录值及其个数；当某个二元组的和为w, 首先在map里寻找map[-w]的值，让答案加上map[-w]，然后再让map[w]++，这样计算后，若存在a + b + c + d = 0, 则一定会被遍历到，且会被遍历到三次，(c, d)的时候遍历到(a, b), (b, d)的时候遍历到(a, c)， (b, c)的时候遍历到(a, d)， 但是会遍历到如a + a + c + d的情况(即一个数被使用了两次)，要将这种情况去除。

多余的答案即是两个不同的数加起来等于另一个其他数的两倍，而这个也是需要两重循环，可以放在一起做。这样得出的答案除3，即为所求。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e7 + 20;

int a[maxn], n;
ll ans;
map<int, int> mp1, mp2;

int main()
{
   FILE *file = fopen("2000.txt", "r");
   fscanf(file, "%d", &n);
   for(int i = 1; i <= n; i++)
        fscanf(file, "%d", &a[i]);
    fclose(file);
   for(int i = 1; i <= n; i++)
       mp2[a[i]]++;

   for(int i = 1; i <= n - 1; i++)
       for(int j = i + 1; j <= n; j++)
       {
           int tmp = a[i] + a[j];
           ans += mp1[-tmp];
           mp1[tmp]++;
           if(!(tmp & 1))
           {
               tmp >>= 1;
               ans -= mp2[-tmp] - (a[i] == -tmp) - (a[j] == -tmp);
           }
       }
    printf("%lld\n", ans / 3);
}
```

用这种算法时，发现10000的数据会超内存，10000的数据理论上内存最大可以是5e7， 可能会爆map。

第二种方法则枚举第二小的数。先将所有数排序，然后枚举第二小的数，这样第一小的数肯定在它左侧，第一大和第二大的数肯定在它右侧，这样用map来维护右侧数的二元组和，然后枚举左侧数，这样就不会重复，若unordered_map的近似度为$O(1)$的话，总体复杂度为$O(n^2)$。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e4 + 20;

int a[maxn], n;
ll ans;
unordered_map<int, int> mp;

int main()
{
   FILE *file = fopen("10000.txt", "r");
   fscanf(file, "%d", &n);
   for(int i = 1; i <= n; i++)
        fscanf(file, "%d", &a[i]);
   sort(a + 1, a + 1 + n);

   mp[a[n]+a[n-1]] = 1;
   for(int i = n - 2; i >= 2; i--)        //枚举第二小
   {
       for(int j = i - 1; j >= 1; j--)     //枚举第一小
       {
           int w = a[i] + a[j];
           ans += mp[-w];
       }
       for(int j = i + 1; j <= n; j++)
           mp[a[i]+a[j]]++;
   }
   printf("%lld\n", ans);
}
```

3. 假设每个数字都与坐标上的点对应，即三个数a,b,c（互不相等）对应$A=(a,a^3),B=(b,b^3),C=(c,c^3)$,则向量$AB=(b-a,b^3-a^3),AC=(c-a,c^3-a^3),A,B,C$共线等价于$(b-a)(c^3-a^3)-(c-a)(b^3-a^3)$，化简得$(b-a)(c-a)(ac-ab+c^2+b^2)=0$,由于$a,b,c$互不相等，可化为$ac-ab+c^2+b^2=0$,即$(a+b+c)(c-b)=0$，则$a+b+c=0$。所以三个数和为0就可化为这三个点共线来做。