a) (3 points) Use pseudocode to specify a brute-force algorithm that determines when given as input a sequence of n positive integers whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of terms of the sequence, checking whether the sum of the first two terms equals the third.

pseudocode:

```
  BRUTE_FORCE(n)                                                               cost
1     result_list=[]                                                           c1
2    for i = 0  to n.length                                                    n
3          for j =i+1 to n.length                                              n-1
4                    for k=j+1 to n.length   #Find all three distinct terms    n-2
5                              if n[i]+[j]=n[k] or n[i]+n[k]=n[j] or n[j]+n[k]=n[i]   c2
6                                        result.append(n[i],n[j],n[k])         c3
7    return result_list
```

b)(2 points) Give a big-O estimate for the complexity of the brute-force algorithm from part (a).

the brute-force algorithm f(n)=O(n^3)

proof:

because lines 2~5
f(n)=O(n*(n-1)*(n-2))=O($n^3 - 3n^2 + 2n$)=O(n^3)

QED.

c)(3 points) Devise a more efficient algorithm for solving the problem that first sorts the input sequence and then checks for each pair of terms whether their sum is in the sequence.

pseudocode:

```
MIN_HEAPIFY(A,i)

  L=2*i

  R=2*i+1

  if L<=A.heap-size and if A[L]<A[i]

        smallest=L

  else smallest=i

  if R<=A.heap-size and if A[R]<A[smallest]

        smallest=R

  if smallest != i

        temp=A[i]
        A[i]=A[smallest]
        A[smallest]=temp

        MIN_HEAPIFY(A,smallest)


BULID_MIN_HEAP(A)

   A.heap-size=A.length

  for i=⌊ A.length/2 ⌋ downto 1

        MIN_HEAPIFY(A,i)

   return A
```

```
HEAPSORT(A)                                      cost

A=BULID_MIN_HEAP(A)                              f2(n)

result_list=[]                                   c1

for i=A.length downto 2                          n-1

     temp=A[1]                                   c2
     A[1]=A[i]                                   c3
     A[i]=temp                                   c4

     result_list.append(A[A.heap-size])          c5

     A.heap-size=A.heap-size - 1                 c6

     MIN_HEAPIFY(A,1)                            f1(n)

return result_list
```

```
Main(A)                                          cost

n=HEAPSORT(A)                                    f3(n)

result_list=[]                                   c1

for i =0 to n.length                             n

     for j= i+2 to n.length                      n-2

          if n[i]+n[i+1]=[j]                     c2

               result_list.append(n[i],n[i+1],n[j])   c3

return result_list
```

run the function Main is the start of all the code

d)(2 points) Give a big-O estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm?

this algorithm f(n)=O(n^2)

proof:

because Main function lines 1~6

f(n)=O(f3(n))+O(n(n-2))

f3(n)=O(f2(n))+(n-1)*O(f1(n))

because
f1(n)<=f1(2n/3)+Θ(1)
Master Theorem T(n)=aT(n/b)+f(n)

==> a=1 b=$\frac{3}{2}$, Θ($n^{\log_b a}$)=Θ(n^0)=Θ(1)
==> T(n)=Θ($n^{\log_b a}\lg n$)=Θ($\lg n$) , f1(n)<=T(n)

==>f1(n)=O($\lg n$)

proof: (the n -element heap 's height $\lfloor \lg n \rfloor$)

because when heap's height =h

==> 2^h <= n <=$2^{h+1}-1$
==> h<=$\lg n$ and $\lg n$ <=h-1 ==> $\lg n$ -1 < h <= $\lg n$
==> h=$\lfloor \lg n \rfloor$

proof: (there are most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height is h in any n-element heap)

because when h=0:
==> the [element n]'s parent node is $\lfloor \frac{n}{2} \rfloor$
==> all the leaves node is n-$\lfloor \frac{n}{2} \rfloor$ =$\lceil \frac{n}{2} \rceil$ — True

hypothesis

when h=x, most nodes is $\lceil \frac{n}{2^{x+1}} \rceil$

when h=x+1, sum the parent nodes of the height h nodes is $\lceil \frac{n}{2^{x+1}} \rceil / 2$

==> $\lceil \frac{n}{2^{x+2}} \rceil = \lceil \frac{n}{2^{(x+1)+1}} \rceil$ — True

so f2(n) = $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^{h}} \right\rceil)$

because $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ when x=1/2

so f2(n)=$O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2})$ ==> <u>f2(n)=O(n)</u>

so f(n)=O(n)+(n-1)*O($\lg n$)+O(n(n-2))

==> f(n)=O(n)+O($\lg n$)+O(n^2)

==> <u>f(n)=O(n^2)</u>

QED.