(11) Check if there are pairs that add up to 0.

```
set<Integer> s = new Hashset<>();
for(int i=0; i< a.length; i++){
    if (s.contains (-1* a[i]))
        return true;
    s.add( a[i]);
}
return false;
```

(12) Get median from running stream of numbers.

- create min and max heap
- ~~add~~ ~~number~~ Rebalance them
- Get median
- Repeat for each element.

```
double[] runningMedians (int[] a){
    PriorityQueue<Integer> lowers = new PriorityQueue<Integer>(
        (Integer a, Integer b) => { return b-a;});

    PriorityQueue<Integer> highers = new PriorityQueue<Integer>();
    double[] medians = new double[a.length];
    for(int i=0; i<a.length; i++){
        if (lowers.size() ==0 || a[i]<lowers.peek())
            lowers.add (a[i]);
        else highers.add (a[i]);

        Rebalance (lowers, highers);

        medians[i] = median( lowers, highers);
    }
    return medians;
}
```

```
void Rebalance (PriorityQueue<Integer> q1, PriorityQueue<Integer> q2){
    if (q1.size() > q2.size() && (q1.size() - q2.size()) >1) {
        q2.offer(q1.poll());
    ~~else if (q2 of q~~
    else if (q2.size() > q1.size() && (q2.size()-q1.size())>1){
        q1.offer( q2.poll());
    }
}

double getMedian (PriorityQueue<Integer> q1, PriorityQueue<Integer>
    q2) {
    if(q1.size() == q2.size())
        return (double)( q1.peek() + q2.peek())/2;

    return q2.size() > q1.size() ? q2.peek() : q1.peek();
}
```

(13) Product Exclusion: For a given array return array where a[i] = prod of all other elements except a[i].

- Get product of all elements ~~of~~ for each element in result divide product by a[i]:

```
int[] prodExclusion (int[] a){
    int product = 1;
    for(int i=0; i<a.length; i++){
        product *= a[i];
    }
    int[] result = new int[a.length];
    for(int i=0; i<a.length; i++){
        result[i] = ~~a[i]~~ product / a[i];
    }
    return result;
}
```