This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write an OpenMP program that uses a Monte Carlo method to estimate π. Read in the total number of tosses before forking any threads. Use a `reduction` clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use `long long ints` for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

5.3. Count sort is a simple serial sorting algorithm that can be implemented as follows:

```
void Count_sort(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
      temp[count] = a[i];
   }

   memcpy(a, temp, n*sizeof(int));
   free(temp);
}  /* Count_sort */
```

The basic idea is that for each element `a[i]` in the list `a`, we count the number of elements in the list that are less than `a[i]`. Then we insert `a[i]` into a temporary list using the subscript determined by the count. There's a slight problem with this approach when the list contains equal elements, since they could get assigned to the same slot in the temporary list. The code deals with this by incrementing the count for equal elements on the basis of the subscripts. If both `a[i] == a[j]` *and* `j <i`, then we count `a[j]` as being "less than" `a[i]`.

After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`.

a. If we try to parallelize the `for i` loop (the outer loop), which variables should be private and which should be shared?

b. If we parallelize the `for i` loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer.

c. Can we parallelize the call to `memcpy`? Can we modify the code so that this part of the function will be parallelizable?

d. Write a C program that includes a parallel implementation of `Count_sort`.

e. How does the performance of your parallelization of `Count_sort` compare to serial `Count_sort`? How does it compare to the serial `qsort` library function?