**1. Solution: Reducing problem to Bineary Search**

Bolt\_Nut\_Match(Bolt, Nut\_width)

{

*# create an empty array to save the matching result*

For i = 1:Nut.length:

arr[i] = 0

*# presort width of Nut array*

 CountSort(Nut)

*# traverse all bolt in Bolt and search its matched Nut*

for i in Bolt:

index = BinearySearch(Nut, i)

arr[i] = Nut[index]

return Bolt, arr

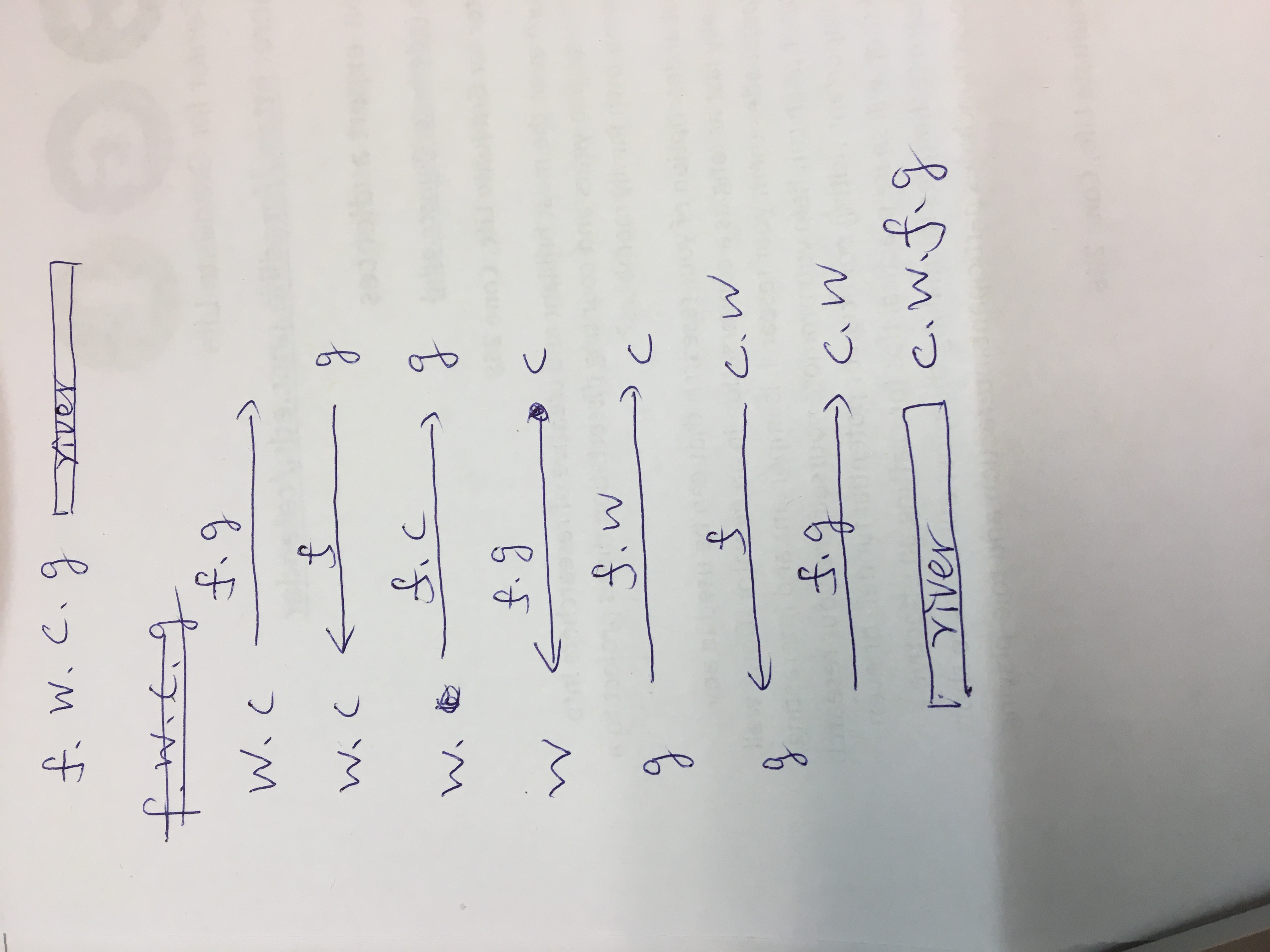
}

**T(n) = (n) + (n) +(nlogn) = (nlogn)**

**2. Solution:** Pick all spaghetti up vertically and make sure all bottom of spaghetti in on the table. We pick up the tallest spaghetti repeatedly. In this way we can sort spaghetti rods by its height in the order from tallest to shortest.

**3. f: farmer w: wolf c: cabbage g: goat**

**Solution is in the next page**



4. **Solution : Reducing problem to linear sort. We can find the median of x-coordinate and this group of coordinate can be the optimize position to minimize the average Manhattan distance from the post office to these points**

FindPosition(A)

{

X = [x for x in A.x] *# A.x represent x coordinate*

Y = [y for y in A.y] *# A.y represent y coordinate*

CountSort(X)

CountSort(Y)

mid\_x = ceil(X/2)

mid\_y = ceil(Y/2)

return (X[mid\_x], Y[mid\_y])

}

The running time of the algorithm can be reduce to O(n).

**5. Short Summary for “Modern B Tree”**

We can understand this paper as a tutorial briefly reviews the basics but assumes that the reader is interested in more detailed information about modern B-tree techniques. The tutorial focuses on duplicate key values (including bitmaps, column storage, and compression), updates (including alternative B-tree structures, load utilities, and update execution plans), and the effects of novel

hardware (including very large memory, flash storage, and memory hierarchies).

**Firstly, in duplicates issues**, the author introduced some traditional representations first. Then the author introduce two methods: Bitmap B-trees and Column storage B-trees. **For Bitmap B-trees**, Bitmaps in database indexes are a fairly old idea that gained importance with the rise of relational data warehousing. This method need a one-to-one mapping between information associated with index keys and integers. Bitmaps are used primarily for read-only or read-mostly data, not for update-intensive databases and indexes. **For Column storage B-trees**, it offers two benefits over traditional B-tree indexes: namely faster scans due to shorter records and better compression due to a uniform data type and domain. The technique is that in each B-tree page, the page header stores the lowest tag value among all B-tree entries on that page, and the actual tag value for each individual B-tree entry is calculated by adding this value and the slot number of the entry within the page. There is no need to store the tag value in the individual B-tree.

**Secondly, in large updated issues,** the author introduced three strategies: Index by index, Buffered B-trees and Partitioned B-trees. **For Buffered B-trees,** B trees can be divided into two groups. Both groups rely on some form of buffering to delay B-tree maintenance and to gain some economy of scale. **For Partitioned B-trees,** The essence idea of partitioned B-trees is to maintain partitions within a single B-tree and to reorganize and optimize such a B-tree online using, effectively, the merge step well known from external merge sort.

**Finally, the author introduced the effect of hardware.**