Project 3 Report Summary

The purpose of this program 3 was to empirically analyzes the running time of various sorting algorithms. An array was generated with random number then its values were deep copied to two other arrays of the same size. Each array was run through one of the following sorting methods: Quick Sort; Merge Sort; and a non-recursive, semi-in-place version of the Merge Sort called Merge Improved. The program did this repeatedly with arrays from size 20 to 1,000, increasing the size by 20 each iteration (20, 40, ... 980, 1000). The start time and end time of each sorting method was recoded, and the overall elapsed time was written to a "compare.txt text" document. This document reports the findings from this program.

Comparison of Data From the Sorting Methods

Figure 1 illustrates the data from "compare.txt." Quick sort was the fastest to complete sorting through an array followed by merge improved, finally merge sort took the longest eclipsing the others. The merge sort is much slower due to array copying operations at each recursive call. Merge improve solves this issue by being semi-in-place and non-recursive.

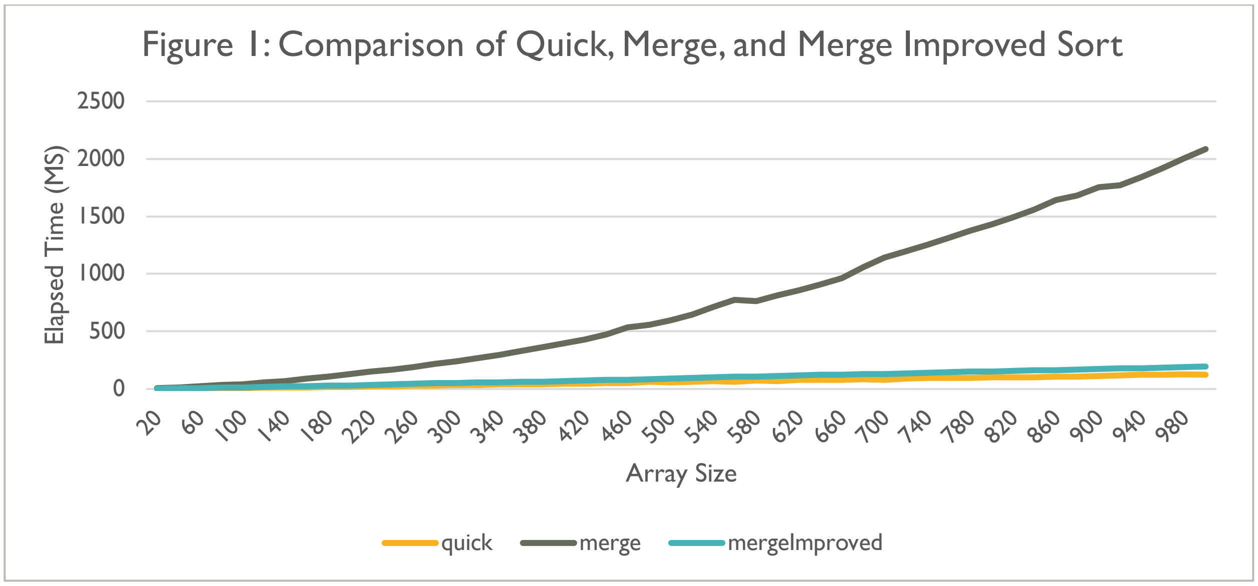
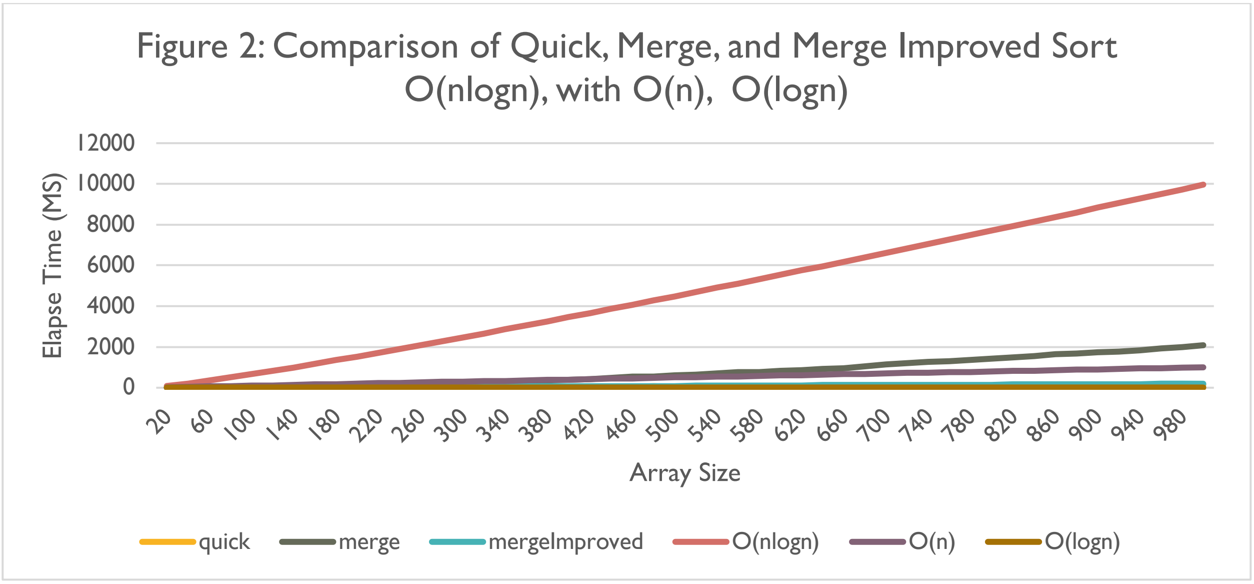
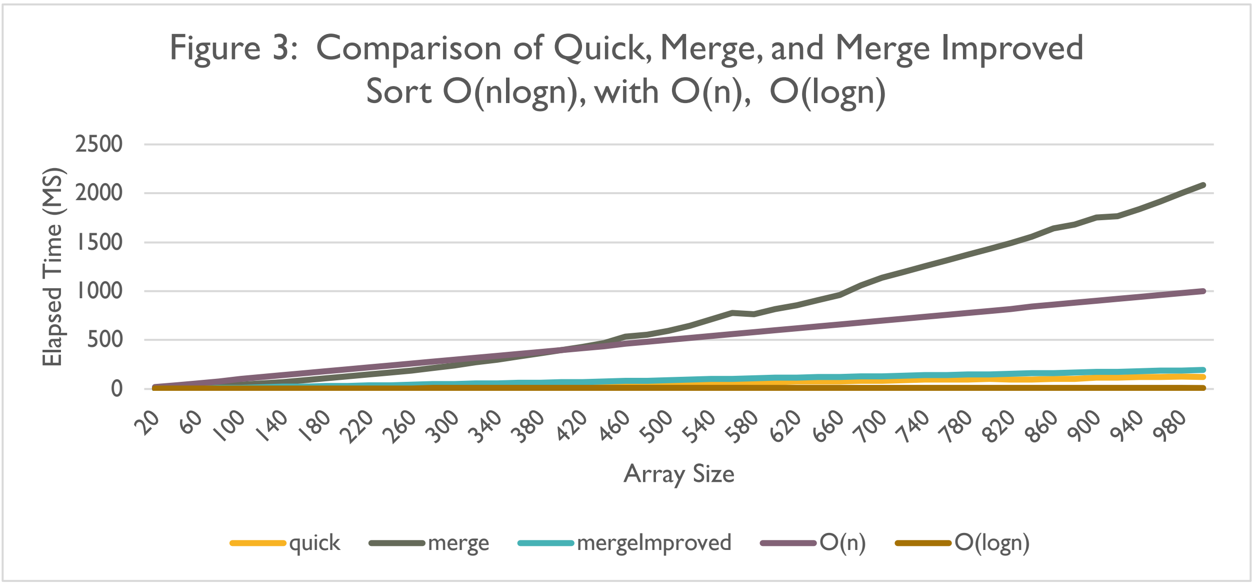


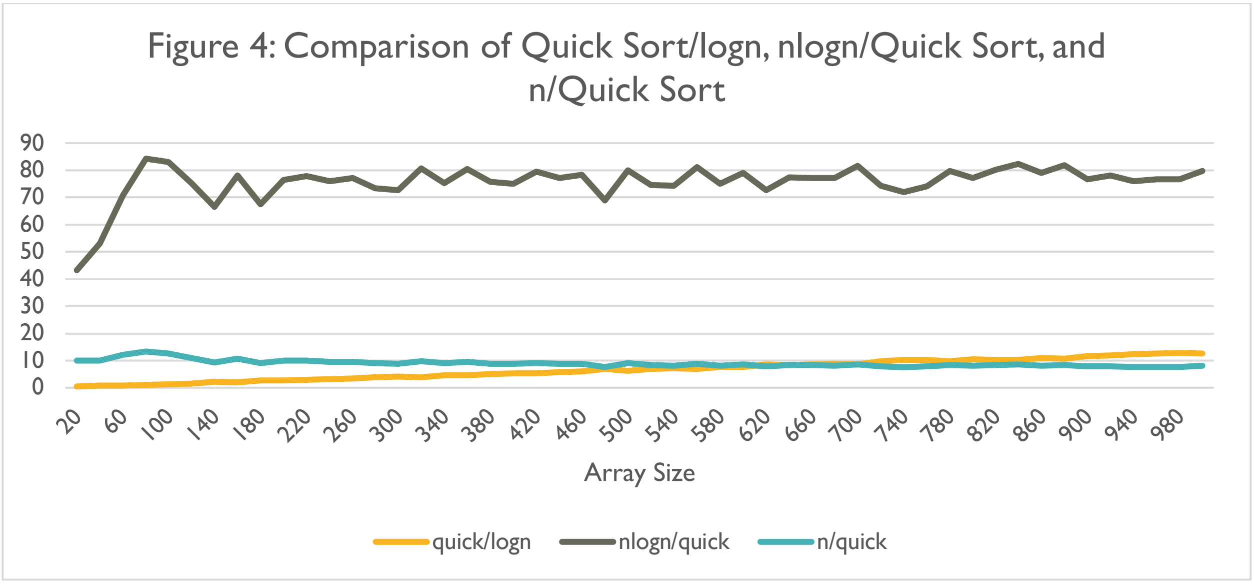
Figure 2 shows the data from "compare.txt" along with the graphs nlog(n), log(n), and n where n is the size of the array. In figure 3 the graph for nlog(n) has been removed to better see the other values.



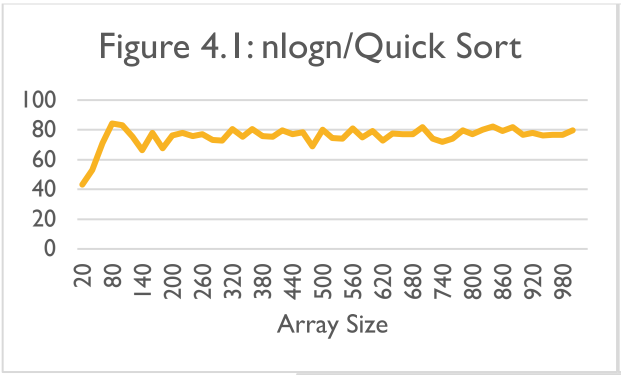


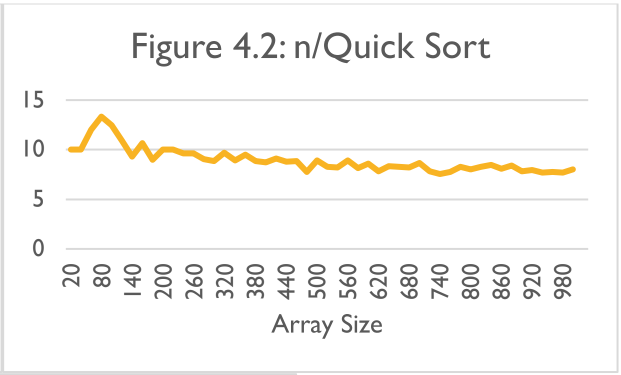
Algorithm Analysis for Qucik Sort

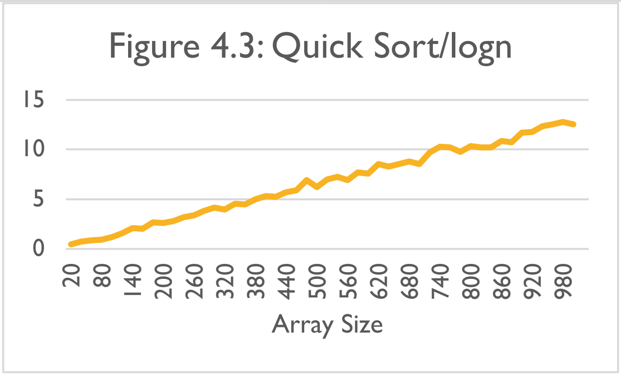
Looking at figure 2 and figure 3 quick sort's upper bounds are nlog(n) and n. It's the lower bound is log(n). The time complexity of quick sort can be proved by dividing the algorithm time and the boundaries then comparing the ratios to find a constant trend. Figure 4 illustrates the ratios you receive when dividing by the appropriate values. The figure shows that dividing nlog(n)/quick results in a constant line, while n/quick results in a decreasing linear function and quick/log(n) results in an increasing linear function.



To better illustrate these trends figures 4.1, 4.2, and 4.3 separate each graph to better examine the pattern. It is easier to see that n/quick(figure 4.2) is decreasing and quick/logn(figure 4.3) is increasing.



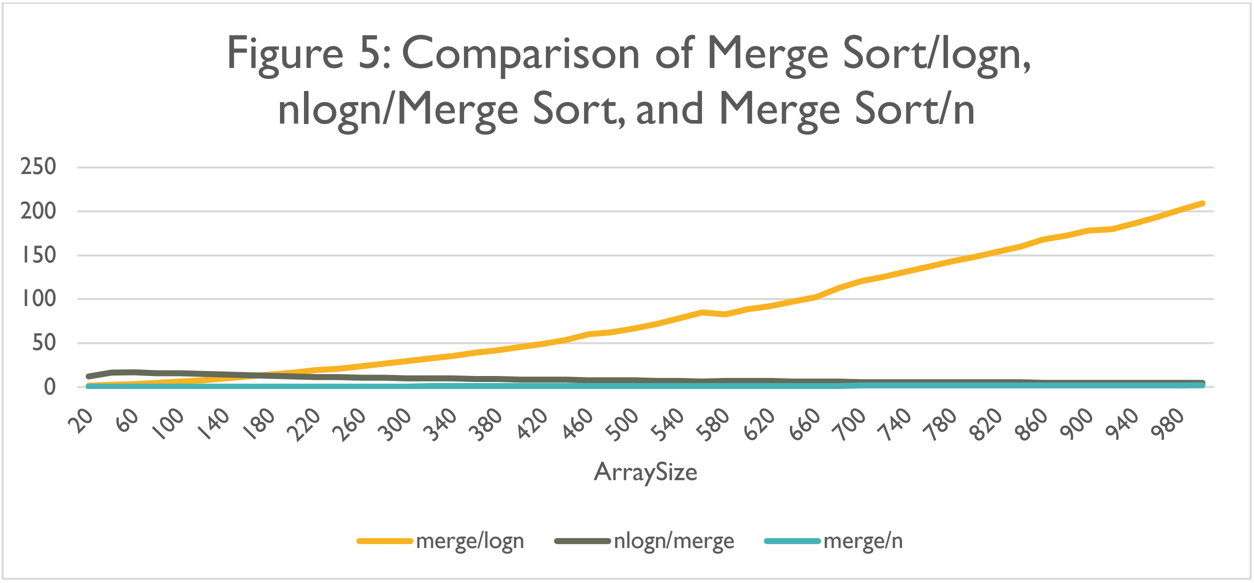




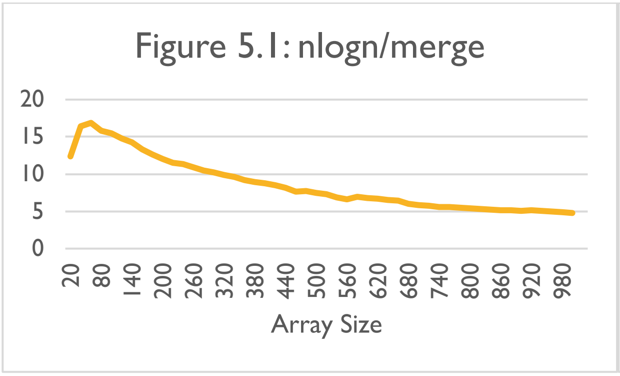
Since dividing quick sort by nlog(n) results in a constant line I can conclude that quick sort has a time complexity of O(nlogn).

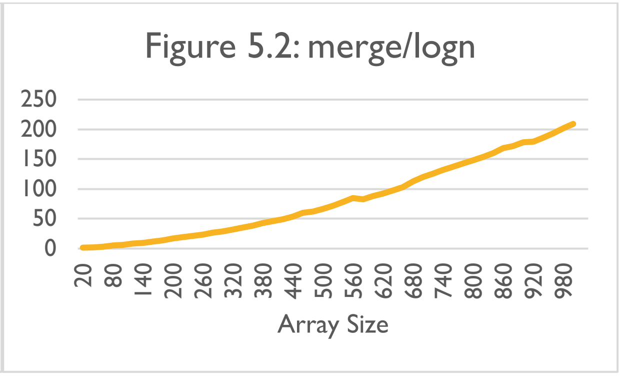
Algorithm Analysis for Merge Sort

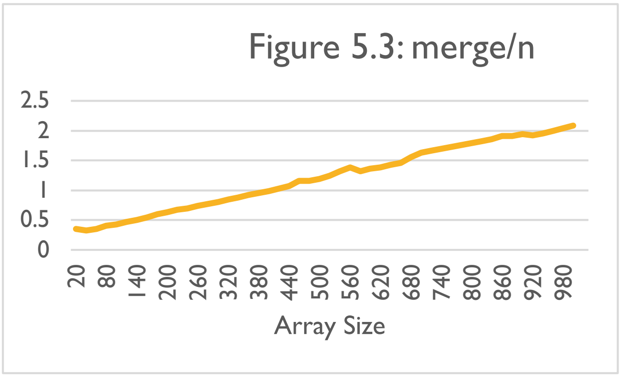
Looking at figure 2 and figure 3 merge sort's upper bounds are nlog(n) and the lower bound is log(n) and n. The time complexity of merge sort can be proved by dividing the algorithm time and the boundaries then comparing the ratios to find a constant trend. Figure 5 illustrates the ratios you receive when dividing by the appropriate values. The figure shows that dividing nlog(n)/merge fluctuates at the beginning but approaches constant as size increase, while merge/logn and merge/n results in an increasing linear function.



To better illustrate these trends figures 5.1, 5.2, and 5.3 separate each graph to better examine the pattern. Here it is easy to see merge/log(n) (figure 5.2) and figure merge/n (figure 5.3) are increasing. While it may look like nlogn/merge (figure5.1) is decreasing it becomes more constant as size increases.



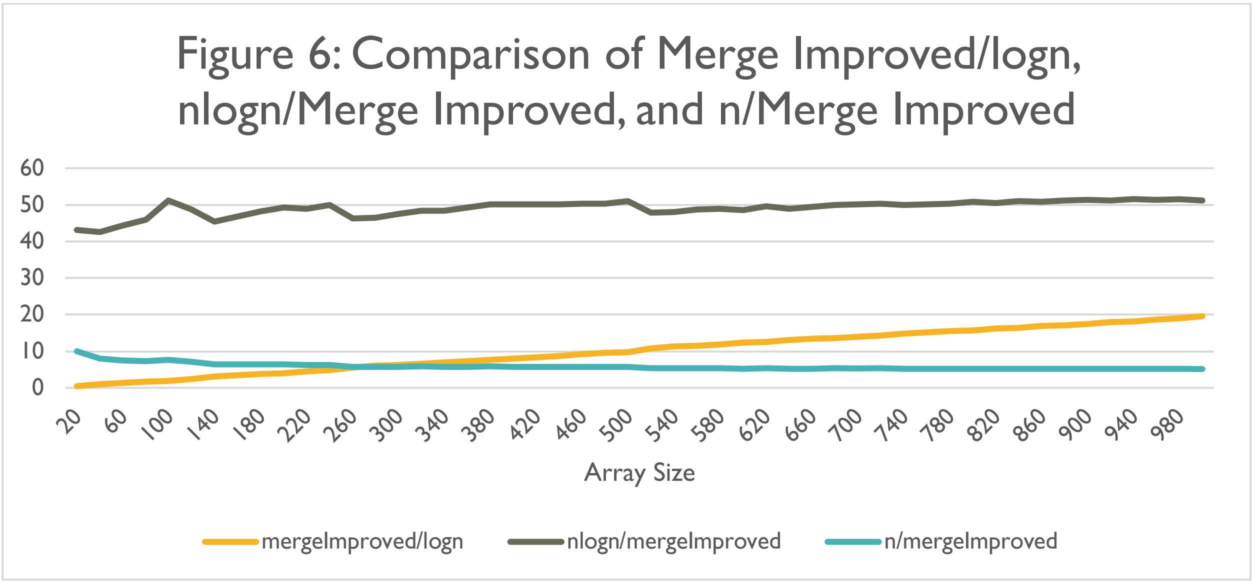




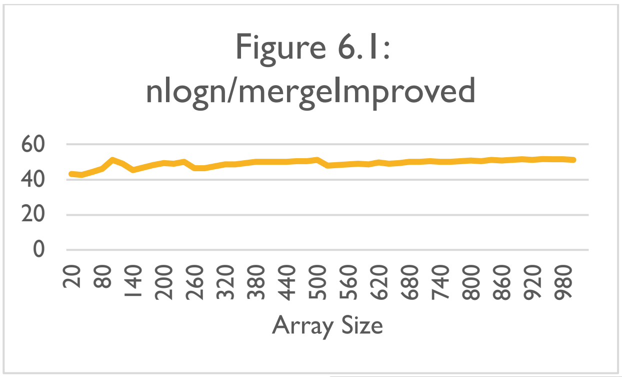
Since dividing merge sort by nlog(n) results in the most constant line I can conclude that merge sort has a time complexity of O(nlogn).

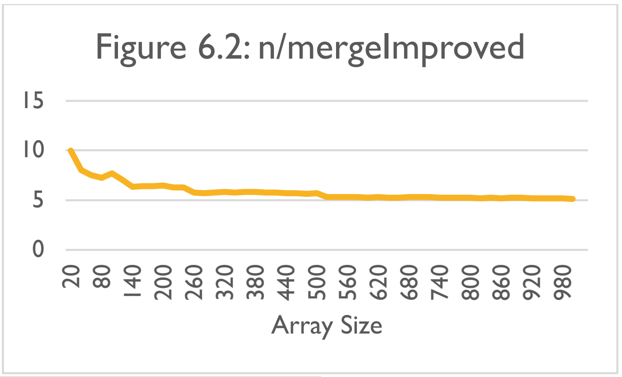
Algorithm Analysis for Merge Improved Sort

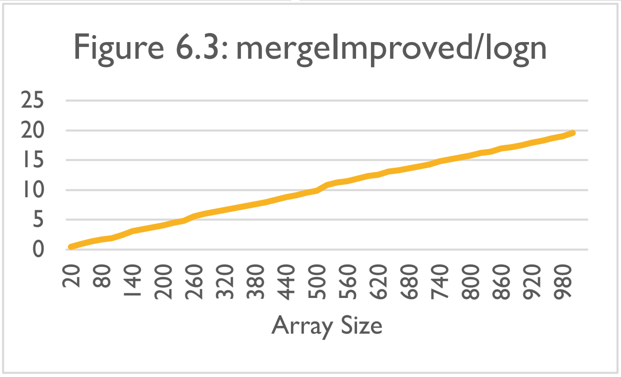
Looking at figure 2 and figure 3 merge improved sort's upper bounds are nlog(n) and n. It's the lower bound is log(n). The time complexity of merge improved sort can be proved by dividing the algorithm time and the boundaries then comparing the ratios to find a constant trend. Figure 6 illustrates the ratios you receive when dividing by the appropriate values. The figure shows that dividing nlog(n)/merge improved results in a constant line, while n/merge improved results in a decreasing linear function and merge improved/log(n) results in an increasing linear function.



To better illustrate these trends figures 6.1, 6.2, and 6.3 separate each graph to better examine the pattern. It is easier to see that n/merge-improved (figure 6.2) is decreasing and merge improved /logn (figure 6.3) is increasing.







Since dividing merge improved sort by nlog(n) results in a constant line I can conclude that merge improved sort has a time complexity of O(nlogn).

Conclustion

From the trends in data and algorithms analysis I have proven the quicksort, merge sort, and merge improved sort have a complexity of O(nlogn). Among all the sorts quick sort was the fastest, followed by merge improved, then merge sort.

Merge improved Output

One-page output of your improved merge sort program (when #items = 30)

initial:	sorted:
items[0] = 13	items[0] = 0
items[1] = 16	items[1] = 1
items[2] = 27	items[2] = 2
items[3] = 25	items[3] = 3
items[4] = 23	items[4] = 4
items[5] = 12	items[5] = 5
items[6] = 9	items[6] = 6
items[7] = 1	items[7] = 7
items[8] = 2	items[8] = 8
items[9] = 7	items[9] = 9
items[10] = 20	items[10] = 10
items[11] = 19	items[11] = 11
items[12] = 0	items[12] = 12
items[13] = 6	items[13] = 13
items[14] = 22	items[14] = 14
items[15] = 11	items[15] = 15
items[16] = 8	items[16] = 16
items[17] = 29	items[17] = 17
items[18] = 18	items[18] = 18
items[19] = 3	items[19] = 19
items[20] = 21	items[20] = 20
items[21] = 14	items[21] = 21
items[22] = 5	items[22] = 22
items[23] = 26	items[23] = 23
items[24] = 15	items[24] = 24
items[25] = 17	items[25] = 25
items[26] = 24	items[26] = 26
items[27] = 10	items[27] = 27
items[28] = 28	items[28] = 28
items[29] = 4	items[29] = 29