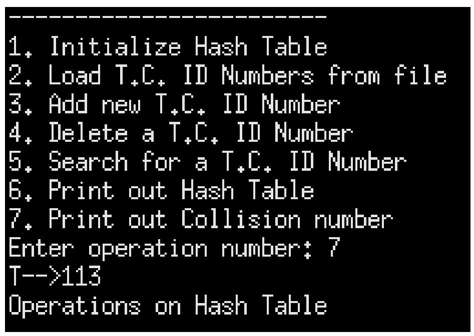
EE 441 DATA STRUCTURES PROGRAMMING ASSIGNMENT #3 REPORT

Mustafa Bıyık
2231454
METU

Hashing Question

First initialize hash table(otherwise it gives segmentation error)(1)
Enter N value("100" as specified in the question)
Select hash function according to specified characters(F,M,T)
Load T.Cs from the file
Then print the collision numbers repeating the same process
F→115
M→178
T→113



Results are following;
1) For loading factor of the each hash functions are equal
because N=100 and we load 200 ID for three cases then loading factor
equal to 2 for three of them.
2) Folding hash function 115 collisions
Truncation hash function 113 collisions
Middle squaring hash function 178 collisions
3) between three of them collision number highest at the Middle squaring hash function
it is expected because we take the square this increase the collision probability

If we want to desire less collision, we can prefer Truncation. If we want to desire a lot of collisions, we can choose Middle Squaring hash function. In practical implementation, less collision is preferred. Therefore, Truncation provides the best result.

Sorting Question

Array size: 100 numbers

Algorithm	#comparisons	#moves	time (msec)
Bubble	4950	2356	130.8 msec
Selection	4851	99	69.4 msec
Quick_1	5050	163	32.2 msec
Quick_2	5050	182	32.4 msec
Quick_3	5050	262	44.6 msec
Quick_4	5050	268	52.6 msec
OWN	5050	160	29 msec

Figure 1:Average #comparisons, #data moves and elapsed time of the 5 arrays of size 100

Array size: 1000 numbers

Algorithm	#comparisons	#moves	time (msec)
Bubble	499500	247637	6236.6 msec
Selection	498501	999	2801.8 msec
Quick_1	500500	2376	244.6 msec
Quick_2	500500	2581	263.8 msec
Quick_3	500500	3439	298.8 msec
Quick_4	500500	3414	330.8 msec
OWN	500500	2313	211.2 msec

Figure 2:Average #comparisons, #data moves and elapsed time of the 5 arrays of size 1000

Array size: 5000 numbers

Algorithm	#comparisons	#moves	time (msec)
Bubble	12497500	6252227	66719.6 msec
Selection	12492501	4999	25560 msec
Quick_1	12502500	14683	505.2 msec
Quick_2	12502500	15597	503.4 msec
Quick_3	12502500	19786	603 msec
Quick_4	12502500	19936	701 msec
OWN	12502500	14455	469.2 msec

Figure 3:Average #comparisons, #data moves and elapsed time of the 5 arrays of size 5000

Array size: 10000 numbers

Algorithm	#comparisons	#moves	time (msec)
Bubble	49995000	24974706	269111 msec
Selection	49985001	9999	102489 msec
Quick_1	50005000	31295	1033.6 msec
Quick_2	50005000	33790	1050.4 msec
Quick_3	50005000	41898	1303.4 msec
Quick_4	50005000	42196	1469.6 msec
OWN	50005000	30794	1032.2 msec

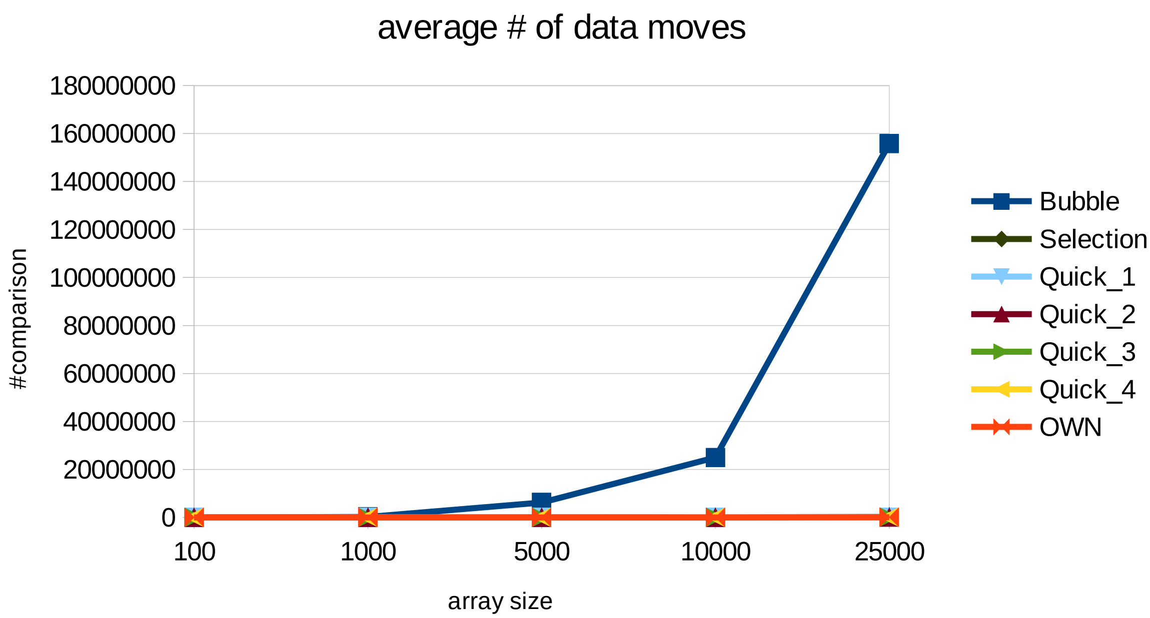
Figure 4:Average #comparisons, #data moves and elapsed time of the 5 arrays of size 10000

Array size: 25000 numbers

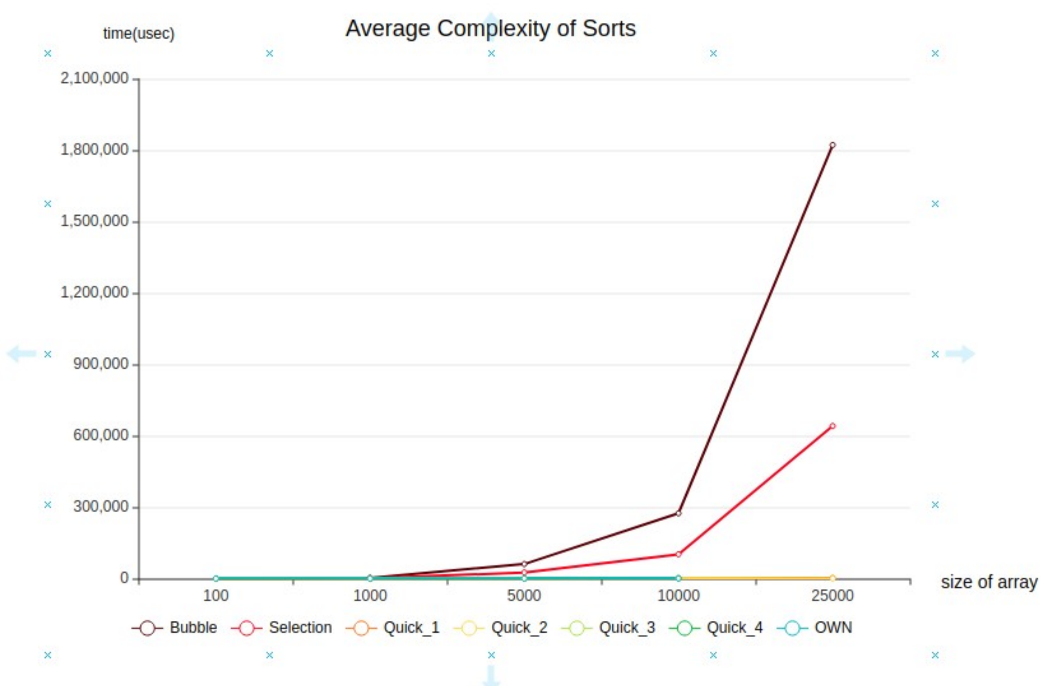
Algorithm	#comparisons	#moves	time (msec)
Bubble	312487500	155800088	1.83047e+06 msec
Selection	312462501	24999	645563 msec
Quick_1	312512500	86328	3013.2 msec
Quick_2	312512500	91912	2891 msec
Quick_3	312512500	112067	3546.8 msec
Quick_4	312512500	113123	3988.8 msec
OWN	312512500	85140	2792.4 msec

Figure 5:Average #comparisons, #data moves and elapsed time of the 5 arrays of size 25000

Quick_1 (select first element as pivot)
Quick_2 (select middle element as pivot)
Quick_3 (select randomly chosen element of the array – array (random_index) – as pivot)
Quick_4 (select the median of 3 randomly chosen elements of the array as pivot)



Graph 1: average number of data moves with respect to array size



Graph 2: average time of the process with respect to array size

MustafaBiyikSort(OWN) is created with Quick_1 and Bubble sort algorithms. Specific cutoff size is declared at the beginning of the Q-2 code. When the array size larger than cutoff size firstly Quick sort algorithm is executed. Then, array is divided into small pieces, this pieces sorted in the modified bubble sort algorithm. Also there is recursion while sorting of the array.

As we can see in the Figure 1, Figure 2, Figure 3, Figure 4 and Figure 5, elapsed time is the least at the OWN algorithm. Why is it the fastest algorithm? Because when the array size is small enough, quick sort runs slower Therefore, overall process is slowed down when we sort the array. Bubble sort more effective while sorting small pieces of the array.

Bubble sort's and selection sort's complexity increase exponentially when we increase the array size. However Quick sort algorithms and OWN algorithm's complexity increases linearly. That makes Quick Sort faster.
When the array is nearly sorted, bubble sort gives complexity n but quick sort gives complexity n^2.
A sorting technique is stable if it does not change the order of elements with the same value. According to this technique bubble sort is stable. Selection sort and Quick sort are unstable.

In the selection sort move #data moves is N-1. This is expected because we find the smallest one and swap. Therefore, number of data moves is the least at the selection sort.

Number of the comparisons according to the sort types

- Bubble =Size*(Size-1)
- Selection=(Size-2)*(Size-1)
- Quick_1=Size*(Size+1)
- Quick_2 =Size*(Size+1)
- Quick_3 =Size*(Size+1)
- Quick_4 =Size*(Size+1)
- if(Size<cutoff){ OWN =Size*(Size-1) }
- else if(Size>=cutoff){ OWN =Size*(Size+1)

- Complexity of the algorithms(elapsed_time)

Bubble>>Selection>>Quick_4>Quick_3>Quick_1>Quick_2>OWN

- #Comparison number of the algorithms

Quick_4=Quick_3=Quick_1=Quick_2>OWN>Bubble>Selection>

- #Data movements of the algorithms

Bubble>>Quick_4>Quick_3>Quick_2>Quick_1>OWN>Selection