Marmara University
Faculty of Engineering

COMPUTER ENGINEERING DEPARTMENT

- HW2-

COURSE LECTURER:

Yrd. Doç. Dr. Ömer KORÇAK

Name Surname: Sezin Gümüş

ID: 150113841

Name Surname: Gülşah Yılmaz

#ID: 150113854

We design an experiment to compare following four sorting algorithms:

1.Quick-sort – Median of three pivot seleciton
2. Switch to insertion-sort in small subarrays
3. Non Recurisively mix all

STEP 1

a) In this step deciding on reasonable inputs and / or generating reasonable sample inputs We want to explore if the median of tree or vanillia quicksort algorithm which sort algorithm is the best? And which sort algorithm is the worst? If the list is randomly sorted which one is the better option for our problem?

In this section we will implement with the coding language c for this purpose we have generated 3 "mixdata.txt" which included randomly genereted sets of numbers quicksort number is between 0- 5000, We are apply in quicksort median of three pivot selection.

Median-of-three pivot selection:

- select leftmost, middle and rightmost element
- order them to the left partition, pivot and right partition. Use the pivot in the same fashion as regular quicksort.

The common pathologies $O(N^2)$ of sorted / reverse sorted inputs are mitigated by this. *It is still easy to create pathological inputs to median-of-three. But it is a constructed and malicious use. Not a natural ordering.*

```
int main()
{
   FILE *fptr =  fopen("mixdata.txt","r");
    int number [5000],i=0;
    clock_t t1,t2;
   if (fptr == NULL)
     printf("File does not exists \n");


    fscanf (fptr, "%ld\n",&number[0]);
    while (!feof (fptr))
     {
       i++;
       fscanf (fptr, "%ld\n",&number[i]);
     }
      fclose(fptr);


     t1 = clock ();
    quickSort(number, 0, 5000 );
     t2 = clock ();
     float sec = (float)(t2-t1)/CLOCKS_PER_SEC;
     printf("Time :%lf",sec);



}
```

For quicksort algorithm implementation is below;

```
        void insertionSort (int  list [],int s, int e)
        {
        int i =0;
          for (i=s+1;i<e;i++)
           {
              int val = list [i];
              int j = i-1;
              while (j>= 0 && val < list[j])
              {
                list [j+1] = list[j];
                j--;
              }
              list[j+1]= val;
           }
        }
```

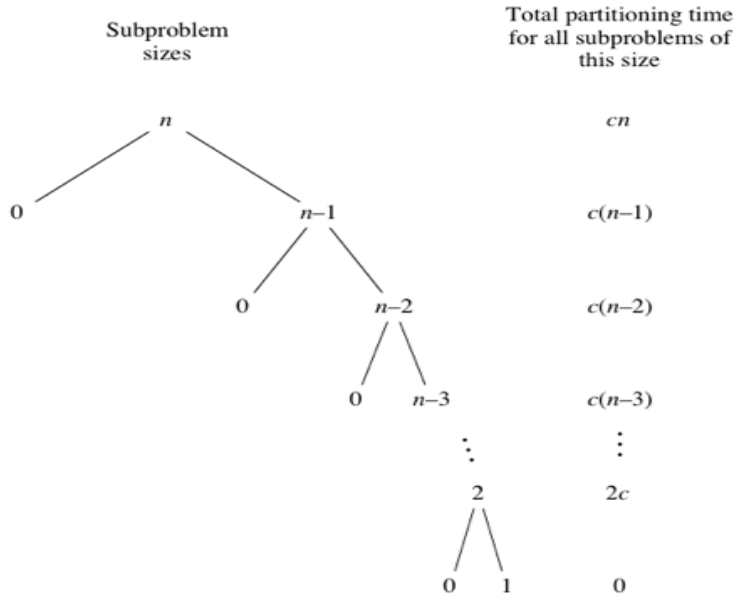
The median of three has you look at the first, middle and last elements of the array, and choose the median of those three elements as the pivot.

```
int partition (int list [] ,int l ,int r)

{
int left = l;
int right = r;
int pivot = list[l];
while (l<r)
  {
   if  (list[left]<pivot)
      left++;
    else if (list[right]<pivot)
        right--;

    int tmp = list[left];
    list[left] = list[right];
    list [right] = tmp;
    left++;
   }
return left;

}
```

This would be better because it reduces the probability of finding "bad" pivots.

The common/**vanilla quicksort** selects as a pivot the rightmost element. This has the consequence that it exhibits pathological performance $O(N^2)$ for a number of cases. In particular the sorted and the reverse sorted collections. In both cases the rightmost element is the worst possible element to select as a pivot. The pivot is ideally thought to me in the middle of the partitioning. The partitioning is supposed to split the data with the pivot into two sections, a low and a high section. Low section being lower than the pivot, the high section being higher.

(Worst-case running time)



b)This time we used median of three and insertion sort algorithm.

```
int main()
{
   FILE *fptr =  fopen("mixdata.txt","r");
    int number [5000],i=0;
    clock_t t1,t2;
   if (fptr == NULL)
     printf("File does not exists \n");


    fscanf (fptr, "%ld\n",&number[0]);
    while (!feof (fptr))
     {
       i++;
       fscanf (fptr, "%ld\n",&number[i]);
     }
      fclose(fptr);


     t1 = clock ();
    quickSort(number, 0, 5000 );
     t2 = clock ();
     float sec = (float)(t2-t1)/CLOCKS_PER_SEC;
     printf("Time---:%lf",sec);
       return 0;


}
```

We apply insertion sort for sub array., idea behind **insertion sort**. Loop over positions in the array, starting with index , need to insert it into the correct place in the sorted subarray to the left of that position. This sort is an improved version of the Bubble Sort algorithm.

Time complexity is O (n ^ 2).

Here our code will implement

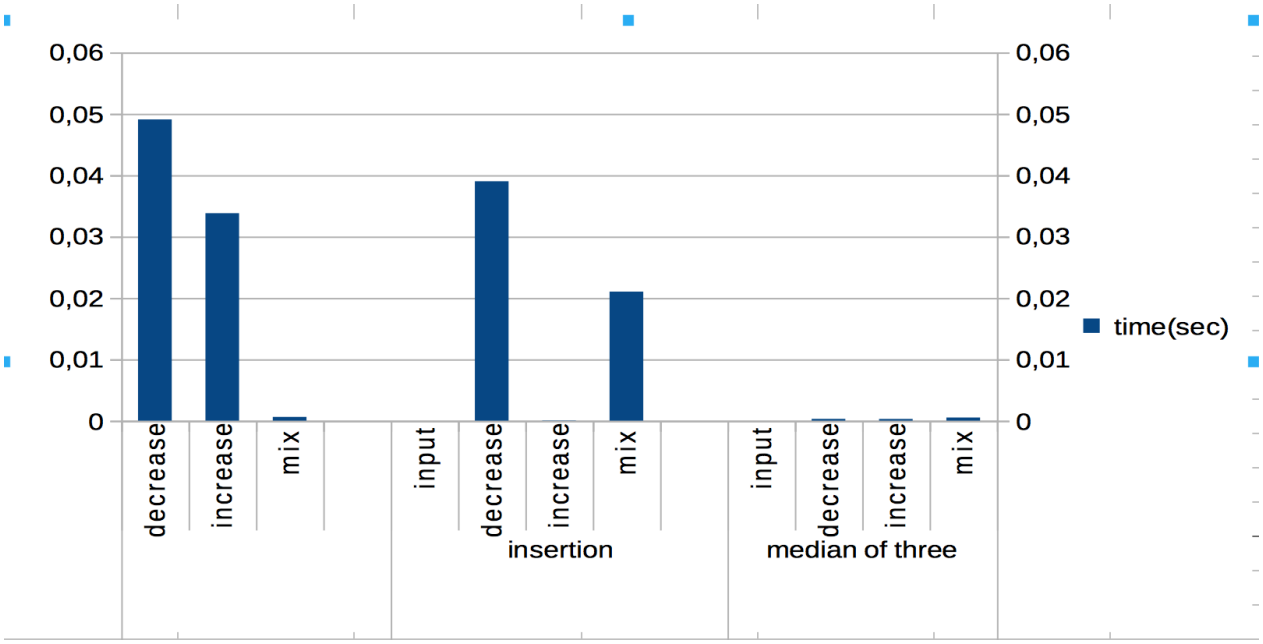
```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
    return ;
}


int median3(int a[], int left, int right)//Uses median of three partitioning technique
{
    int center = (left + right)/2;
    if (a[center] < a[left])
        swap(&a[left],&a[center]);
    if (a[right] < a[left])
        swap(&a[left],&a[right]);
    if (a[right]< a[center])
        swap(&a[center],&a[right]);

    swap(&a[center], &a[right - 1]);//since the largest is already in the right.
    return a[right - 1];
}

void insertionSort (int  list [],int s, int e)
{
int i =0;
  for (i=s+1;i<e;i++)
   {
      int val = list [i];
      int j = i-1;
      while (j>= 0 && val < list[j])
      {
        list [j+1] = list[j];
        j--;
      }
      list[j+1]= val;
   }
}
```

STEP 2

We implemented our programs in C language on computer with two input lists for each sorting algorithms.

STEP 3

a) After generating input files we have run the 4 different sorting algorithms and record the values below:



	Decrease	Increase	Mix
Iterative	0.04907	0.033789	0.000606
Insertion	0.038994	0.000036	0.021012
Median of Three	0.000286	0.000269	0.000501
Insertion sort Median of Three iterative	0.039926	0.000034	0.018901

Sorted List

Insertion sort is better than others in sorted list. The worst is quick-sort(pivot is always selected as the first element). General ranking is insertion sort, quick-sort(median of three), merge and quick-sort(pivot is always selected as the first element) from best to worst.

Reserved Sorted List

Quick-sort(median of three) sort is better than others in reserved sorted list. The worst is insertion sort. General ranking is quick-sort(median of three), merge, quick-sort(pivot is always selected as the first element) and insertion sort from best to worst.

Randomly Sorted List

Quick-sort(pivot is always selected as the first element) sort is better than others in reserved sorted list. The worst is insertion sort. General ranking is quick-sort(pivot is always selected as the first element), quick-sort(median of three), merge and insertion sort from best to worst.

As a result, the best is changing different input list. But insertion sort is generally the worst without sorted list.

	Insertion-Sort	*Quick-Sort*	*Quick-Sort median-of-three pivot selection1.*
Best case	n	nlogn	nlogn
Average case	n	nlogn	nlogn
Worst case	n^2	n^2	nlogn

Quick-sort with median-of-three pivot selection

When the input size doubled:

Theoretically in best case, average case, and worst case should increase 2,16 times. In our experiment best case increases 2,20 times, average case increases 2,58 times and worst case increases 4,20 times. So best case and average case are almost same. But worst case is different than theoretical result.

Quick-sort (pivot is always selected as the first element)

The partition step of quicksort takes n-1 comparisons. So we can write a recurrence for the total number of comparisons done by quicksort:

C(n) = n-1 + C(a) + C(b)

where a and b are the sizes of L1 and L2, generally satisfying a+b=n-1. In the worst case, we might pick x to be the minimum element in L. Then a=0, b=n-1, and the recurrence simplifies to C(n)=n-1 + C(n-1) = O(n^2). So this seems like a very bad algorithm.

0.000286	0.000269	0.000501

When the input size doubled:

Theoretically best=average=worst=2,16

➔ Best case

Theoretically nlogn=2,16. In our experiment 0.000269 . They are not even close.

➔ Average case

Theoretically nlogn=2,16. In our experiment 0.000286 . So they are not even close.

➔ Worst case

Theoretically n^2=4. In our experiment 0.000501 . They are not even closed

Insertion-Sort

➔ Best Case

We measured rate of best case performance 0.000036 . On the other hand theoretically when the input size doubled, execution time should be doubled. We observed that empirical result close to theoretical result.

➔ Average Case

If the input size doubled theoretically execution time increase 5 times. We measure it 0.021012 empirically. And we saw that they are too close.

➔ Worst case

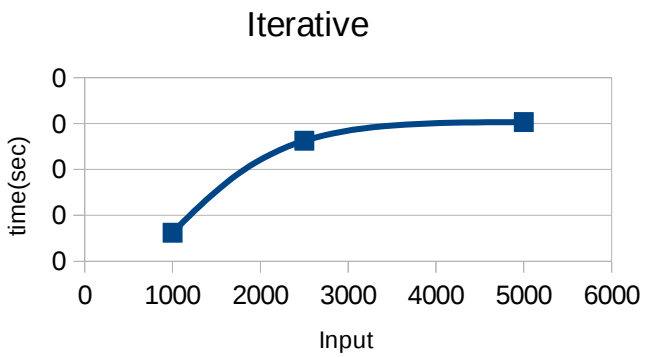
If the input size doubled theoretically execution time should be doubled. And we measure it 0.038994 .It is not very close like best and average case.

Plots by Input Size

In this step , we will see the graph according to input size:

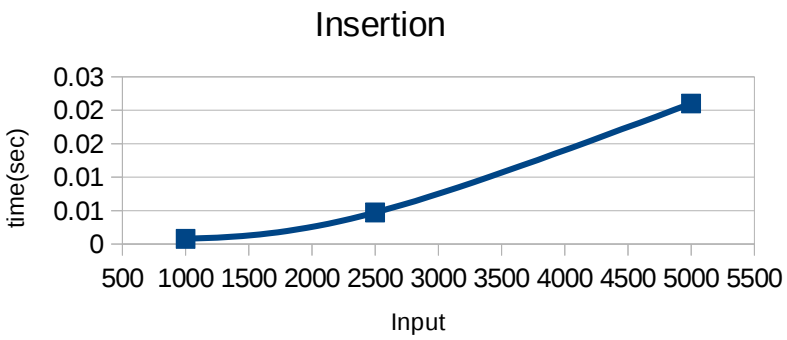
Iterative Quick-sort

Iterative	input size	time(sec)
	5000	0.000606
	2500	0.000525
	1000	0.000124



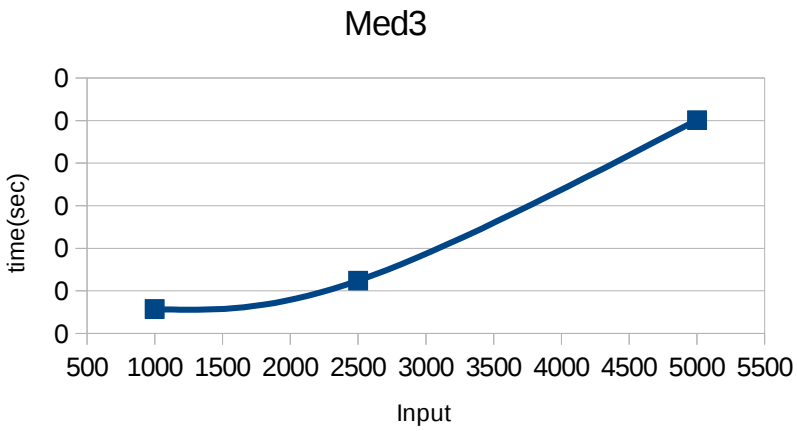
Insertion Quick-sort

	input size	time(sec)
insertion	5000	0.021012
	2500	0.004718
	1000	0.000771



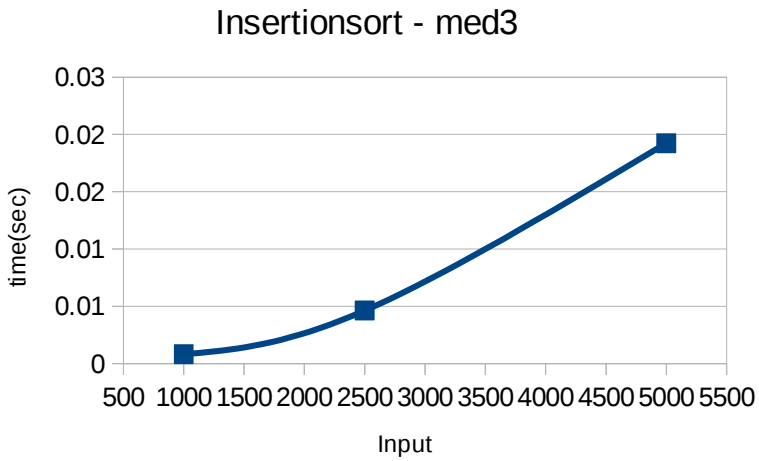
Median-of-three

	input size	time(sec)
med3	5000	0.000501
	2500	0.000124
	1000	0.000057



Insertion Sort – Median-of-three

Insertion-med3	input size	time(sec)
	5000	0.019235
	2500	0.004625
	1000	0.000814



Insertion Sort – Median-of-three – Iterative

Insertion-med3-iterative	input size	time(sec)
	5000	0.018901
	2500	0.004605
	1000	0.00076