BLG381E

Advanced Data Structures

2012 Fall

Report of Project 2

Date of Submission : 16.11 2012

Student Name: Volkan İlbeyli

Student Number : 040100118

Instructor : Zehra Çataltepe

Analysis of sorting algorithms

Heapsort

```
template <class T>
void max_heapify(T* A, int i, int size){
	int l = 2*i + 1;
	int r = 2*i + 2;
	T largest, temp;

	if(l <= size && A[l] > A[i])
		largest = l;
	else
		largest = i;

	if(r <= size && A[r] > A[largest])
		largest = r;

	if(largest != i){
		temp = A[i];
		A[i] = A[largest];
		A[largest] = temp;
		max_heapify(A, largest, size);
	}

	return;
}
```

Since there is no loop in the code, until the recursive call of max_heapify(), the running time of each line is θ(1). By using the Master theorem's case 2, the running time of max_heapify() is θ(*lg*n).

```
template <class T>
void build_max_heap(T* A, int heapsize){
	for(int i = heapsize/2 ; i >= 0 ; i--)
		max_heapify(A, i, heapsize);
	return;
}
```

Running time of build_max_heap() is O(n) since nearly 50% of the elements are leaves and most of the elements are near leaves.

```
template <class T>
void heapsort(T* A, int size){
	T tmp;

	build_max_heap(A, size);
	for(int i = size ; i >= 1 ; i--){
		tmp = A[0];
		A[0] = A[i];
		A[i] = tmp;
		size--;
		max_heapify(A, 0, size);
	}
}
```

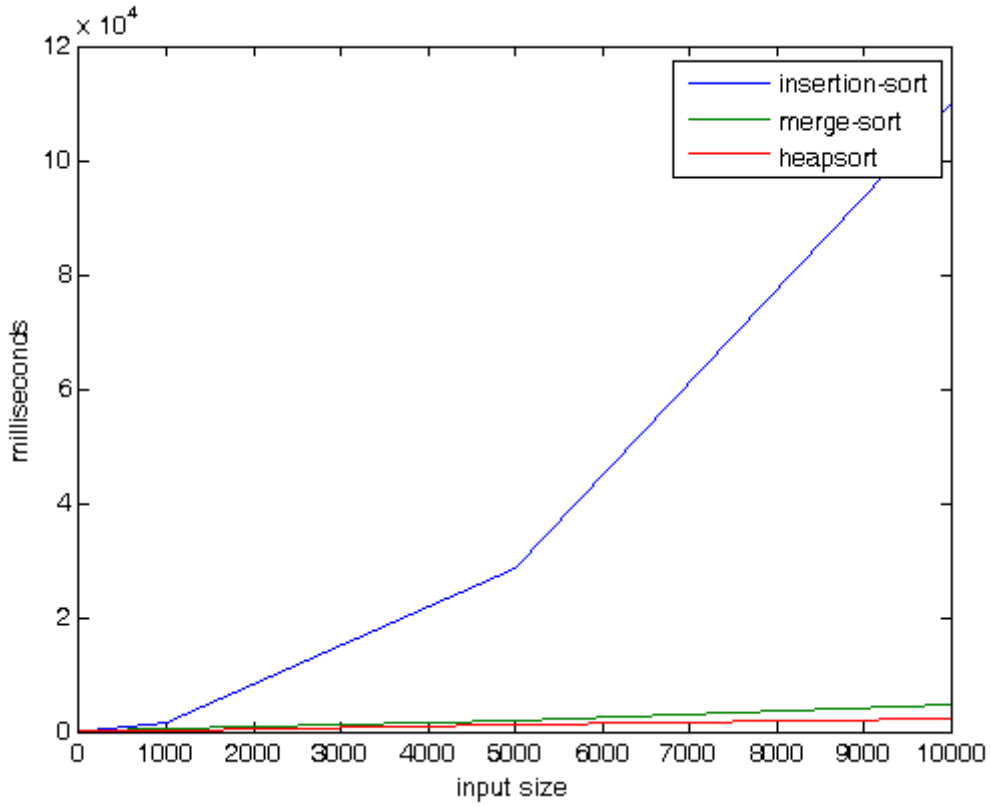
We know max_heapify takes O(*lg*n) time to execute and the loop runs n-1 times. Thus, the running time of heapsort() is n*lg*n.

Heapsort, Merge Sort, Insertion Sort

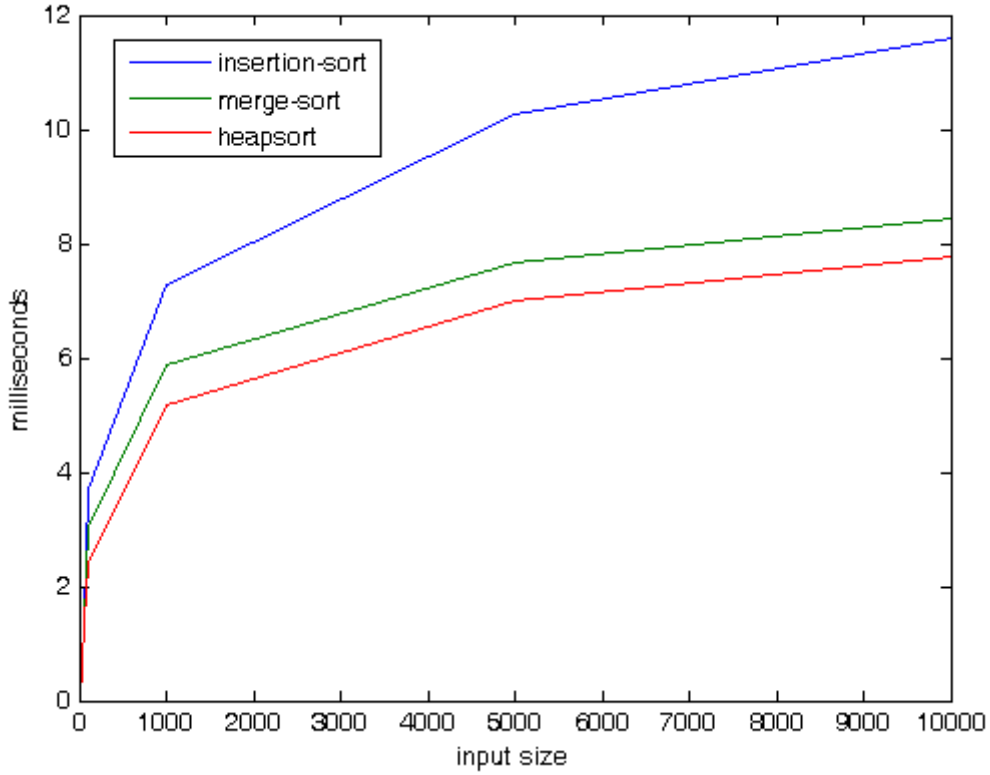
It is known that the insertion-sort algorithm has a complexity of $O(n^2)$ while the merge-sort and heapsort algorithms have $O(n\lg n)$. During the execution of the program, the average time needed for an algorithm to execute is computed. Below is the table that consists running times of the algorithms and the input size which can also be seen from the program output.

Input Size	10	20	50	100	1000	5000	10000
Heapsort	0 ms	1 ms	4 ms	11 ms	174 ms	1100 ms	2400 ms
Merge Sort	0 ms	1 ms	4 ms	21 ms	354 ms	2100 ms	4600 ms
Insertion Sort	0 ms	1 ms	4 ms	41 ms	1434 ms	28600 ms	109900 ms

Since the input size increases dramatically, its impossible to compare merge sort and heapsort. But its possible to understand that they have both the same complexity. Since insertion sort has n^2 comoplexity, its running time takes very long time compared to other two which have a complexity of $n\lg n$.



That is why the logarithm graph is given below:



Now we can clearly see how functions grow.

Since this graph values are generated through a program without a perfect, ideal running-time-computing-algorithm in addition to the values being mean values and having a standard deviation, the graph may not look like the theoretical comparison of sorting algorithms. Even though, the above data is not completely invalid. When we compare the three algorithms, it is observed that heapsort comes handy compared to other 2 algorithms while insertion sort becomes inefficient as the input size increases. I think it is for merge sort uses some extra space associated with its size is why heapsort runs faster than merge sort.

Storage

We know that heapsort and insertion sort are “in place” sorting algorithms which means they require no extra space except for the dummy variables which means no other arrays are created associated with the given array. Thats why they have a $\theta(1)$ space complexity.

On the other hand, merge sort requires extra arrays to merge the sub-arrays which are basically the portions of the original array. Thats why merge sort has $O(n)$ space complexity.

Apparently, heapsort is the best algorithm among the 3 with the shortest running time and having an in place sorting. That is why I would prefer heapsort algorithm if I were to write code for an embedded system where I have storage limitations.