CS202: Fundamentals of Computer Science II

HOMEWORK - 1

Algorithm Analysis and Sorting

Zeynep Begüm Kara - 22003880

CS202 - 01

March 13, 2023

1. Show that $f(n) = 6n^4 + 9n^2 - 8$ is $O(n^4)$.

   Function $f(n)$ is order of n^4 since there exists positive constants c and n_0 such that

   $$f(n) \leq c \cdot n^4 \quad when \quad n \geq n_0$$

   Choose $c = 7$ and $n_0 = 3$

   $$6n^4 + 9n^2 - 8 \leq 7n^4$$

   $$0 \leq n^4 - 9n^2 + 8$$

   $$0 \leq (n^2 - 8)(n^2 - 1)$$

   Since $(n^2 - 8)$ and $(n^2 - 1)$ is always positive when $n \geq 3$,

   $$f(n) \leq 7 \cdot n^4 \quad when \quad n \geq 3.$$

   Therefore, $f(n) = O(n^4)$.

2. Trace of the given array `[ 5 3 2 6 4 1 3 7 ]`

   (a) Selection Sort

   Let us denote subarrays with a vertical bar as follows

   $$Unsorted \mid Sorted$$

   Observe that everything is in unsorted subarray initially. Then, we have the following steps

   ```
   5 3 2 6 4 1 3 7 |
   5 3 2 6 4 1 3 | 7
   5 3 2 3 4 1 | 6 7
   1 3 2 3 4 | 5 6 7
   1 3 2 3 | 4 5 6 7
   1 3 2 | 3 4 5 6 7
   1 2 | 3 3 4 5 6 7
   1 | 2 3 3 4 5 6 7
   | 1 2 3 3 4 5 6 7
   ```

   (b) Merge Sort

   Let us represent arrays which will be merged in next step with `red` or (`red` and `blue`) and resulting merged array with `blue`.

   ```
   [ 5 3 2 6 4 1 3 7 ]
   [ 5 3 2 6 ] [ 4 1 3 7 ]
   [ 5 3 ] [ 2 6 ] [ 4 1 3 7 ]
   [ 5 ] [ 3 ]  [ 2 6 ] [ 4 1 3 7 ]
   [ 3 5 ] [ 2 6 ] [ 4 1 3 7 ]
   [ 3 5 ] [ 2 ] [ 6 ] [ 4 1 3 7 ]
   ```

```
[ 3 5 ] [ 2 6 ] [ 4 1 3 7 ]
[ 2 3 5 6 ] [ 4 1 3 7 ]
[ 2 3 5 6 ] [ 4 1 ] [ 3 7 ]
[ 2 3 5 6 ] [ 4 ] [ 1 ] [ 3 7 ]
[ 2 3 5 6 ] [ 1 4 ] [ 3 7 ]
[ 2 3 5 6 ] [ 1 4 ] [ 3 ] [ 7 ]
[ 2 3 5 6 ] [ 1 4 ] [ 3 7 ]
[ 2 3 5 6 ] [ 1 3 4 7 ]
[ 1 2 3 3 4 5 6 7 ]
```

(c) Quick Sort

Let us define subarrays **S1**, *consists of elements that are strictly smaller than pivot* **S2**, *consists of elements that are greater than or equal to pivot* and **Unsorted**. Denote them as follows

$$S1 \mid S2 \mid Unsorted$$

Recall that pivot will be choosen as last element, denoted in the array with brackets [pivot]. According to our algorithm, we will place [pivot] in first place of the array by swapping. Pivot is denoted as `red` after it is placed according to partitioning function. Observe that everything is in unsorted subarray initially. We have the following steps

```
5 3 2 6 4 1 3 7

[7] | | 3 2 6 4 1 3 5
[7] 3 | | 2 6 4 1 3 5
[7] 3 2 | | 6 4 1 3 5
[7] 3 2 6 | | 4 1 3 5
[7] 3 2 6 4 | | 1 3 5
[7] 3 2 6 4 1 | | 3 5
[7] 3 2 6 4 1 3 | | 5
[7] 3 2 6 4 1 3 5 | |
```

when unsorted subarray is empty place the pivot between S1 and S2 by swapping it with the very last element of S1. Hence,

```
3 2 6 4 1 3 5 7
```

Then, do recursive calls partition function for S1 and S2

```
 3 2 6 4 1 3 5

[5] | | 2 6 4 1 3 3
[5] 2 | | 6 4 1 3 3
[5] 2 | 6 | 4 1 3 3
```

```
[5] 2 4 | 6 | 1 3 3
[5] 2 4 1 | 6 | 3 3
[5] 2 4 1 3 | 6 | 3
[5] 2 4 1 3 3 | 6 |
```

when unsorted subarray is empty place the pivot between S1 and S2 by swapping it with the very last element of S1. Hence,

```
3 2 4 1 3 5 6
```

Keep continuing to these recursive calls, we have following steps for each sort pass

```
5 3 2 6 4 1 3 7
3 2 6 4 1 3 5 7
3 2 4 1 3 5 6 7
2 1 3 4 3 5 6 7
1 2 3 4 3 5 6 7
1 2 3 4 3 5 6 7
1 2 3 3 4 5 6 7
1 2 3 3 4 5 6 7
1 2 3 3 4 5 6 7
```

3. Find asymptotic running time of $T(n) = 2T(n-1) + n^2$ where $T(1) = 1$.

Since $T(n) = 2T(n-1) + n^2$ (*),

$$T(n-1) = 2T(n-2) + (n-1)^2$$

is also true.Replacing $T(n-1)$s in (*), we have

$$T(n) = 2(2T(n-2) + (n-1)^2) + n^2$$

Rearranging terms, it equals

$$T(n) = 2^2T(n-2) + 2(n-1)^2 + n^2 \qquad (**)$$

Similarly, again since (*),

$$T(n-2) = 2T(n-3) + (n-2)^2$$

holds. Replacing $T(n-2)$s in (**), we have

$$T(n) = 2^2(2T(n-3) + (n-2)^2) + 2(n-1)^2 + n^2$$

Rearranging terms, it equals

$$T(n) = 2^3T(n-3) + 2^2(n-2)^2 + 2(n-1)^2 + n^2$$

Let us continue this substitution process. Then, for a $k < n$, the equation becomes

$$T(n) = 2^kT(n-k)+2^k(n-k)^2+2^{k-1}(n-(k-1))^2+...+2^2(n-2)^2+2(n-1)^2+n^2$$

which equals

$$T(n) = 2^k T(n-k) + \sum_{i=0}^{k} 2^i (n-i)^2$$

For $k = n-1$, we have the base case $T(1) = 1$. Hence,

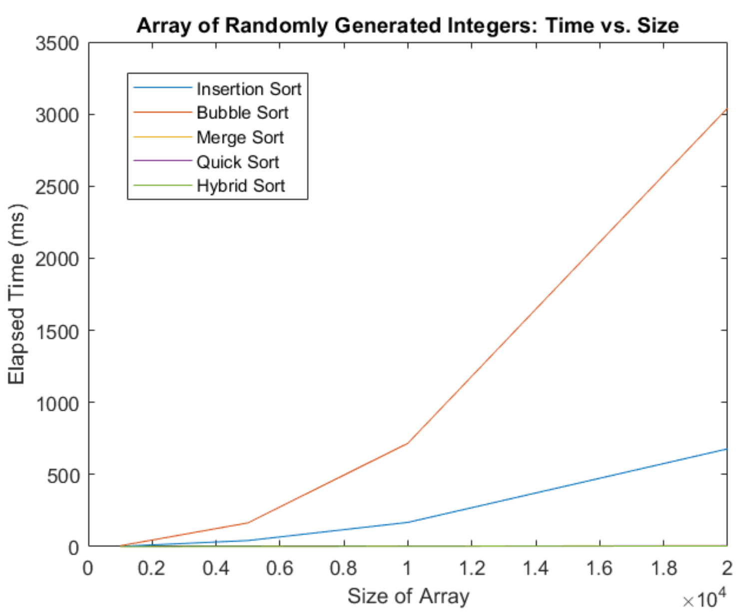
$$T(n) = 2^{n-1} + \sum_{i=0}^{n-1} 2^i (n-i)^2$$

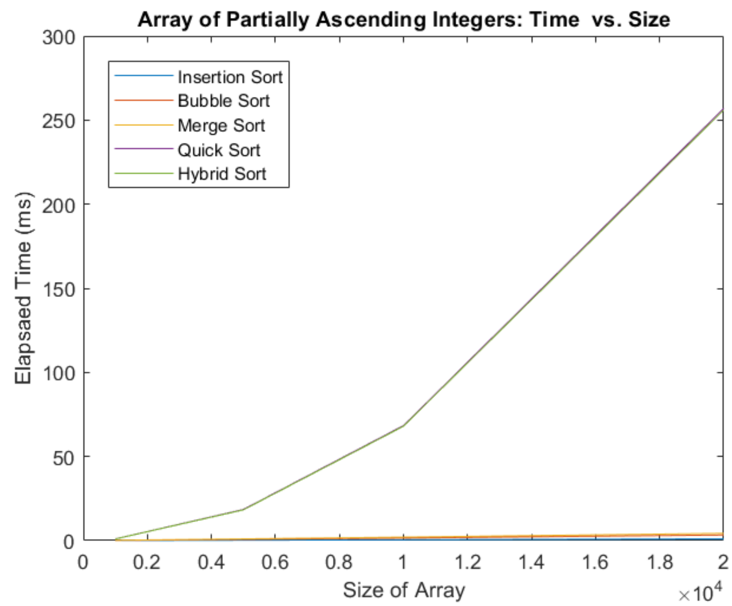
Notice that

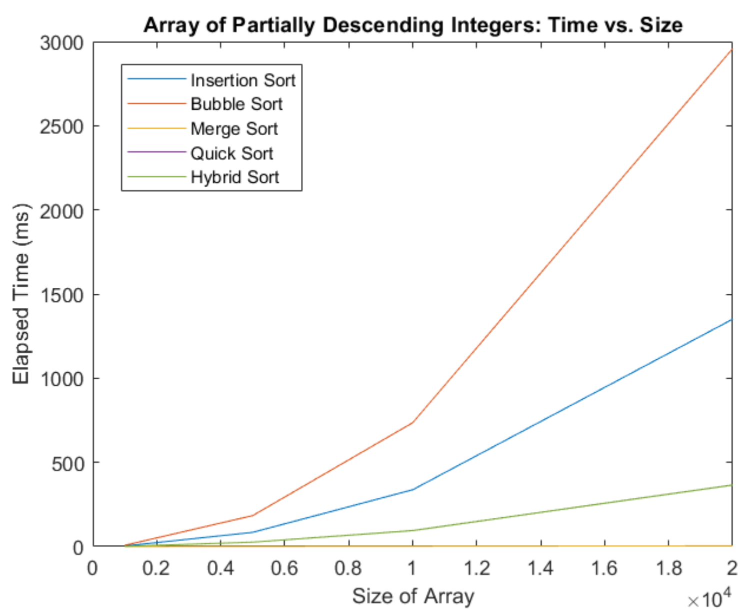
$$T(n) < \sum_{i=0}^{n} 2^i (n-i)^2 < \sum_{i=0}^{n} 2^i n^2 = n^2 \sum_{i=0}^{n} 2^i = n^2(2^{n+1}-1) < n^2 2^{n+1}$$

Since $n^2 2^{n+1} < c \cdot 3^n$ where $n > n_0 = 5$ and $c = 2$, $T(n)$ is order of 3^n.

4. **Performance Plots of Sorting Algorithms for Different Array Characteristics**




Array of Partially Ascending Integers: Time vs. Size
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Hybrid Sort
Elapsaed Time (ms)
Size of Array
×10^4
0
50
100
150
200
250
300
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Array of Partially Descending Integers: Time vs. Size
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Hybrid Sort
Elapsed Time (ms)
Size of Array
×10^4
0
500
1000
1500
2000
2500
3000
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

Array	Elapsed Time (ms)					Number of Comparisons					Number of Data Moves				
	Insrtn Sort	Bubble Sort	Merge Sort	Quick Sort	Hybrid Sort	Insrtn Sort	Bubble Sort	Merge Sort	Quick Sort	Hybrid Sort	Insrtn Sort	Bubble Sort	Merge Sort	Quick Sort	Hybrid Sort
R1K	1.76	5.803	0.246	0.201	0.144	256989	499121	8703	103364	8889	257995	767993	19951	20158	14155
R5K	42.117	164.712	1.393	1.201	0.886	6211279	12490478	55190	69651	61165	6216281	18618851	123615	106990	76033
R10K	168.094	717.251	2.969	2.643	2.003	24877133	49986484	120544	154416	136956	24887144	74601440	267231	260435	199073
R20K	678.179	3042.63	6.315	5.687	4.432	100337977	199988346	260971	343361	308699	100357983	300953957	574463	561811	437668
A1K	0.044	0.104	0.174	1.003	0.958	2709	8954	5760	191924	191652	3709	5135	19951	5425	3357
A5K	0.235	0.687	1.014	18.501	18.3	17587	59921	35613	3824747	3821814	22588	37772	123615	30819	16508
A10K	0.479	1.491	2.051	68.375	68.009	37369	129908	76787	14275756	14268840	47369	82115	267231	63881	32842
A20K	1	3.324	4.309	256.968	256.006	80219	299879	164179	53793880	53778357	100219	180665	574463	131651	65360
D1K	3.383	7.379	0.168	1.348	1.332	497947	499489	5629	140546	140018	499205	1491623	19951	226171	223361
D5K	84.337	183.817	0.96	26.21	26.054	12486787	12497484	34553	2761811	2757592	12492893	37448687	123615	4663142	4645685
D10K	337.453	735.683	2.04	94.992	94.547	49979332	49994998	75108	10029830	10020144	49982410	149887238	267231	16952926	16914958
D20K	1350.97	2958.25	4.28	365.829	365.191	199935915	199989996	160974	38652981	38632900	199959943	599759837	574463	65324661	65246008

Results

In this study, it is aimed to observe the behaviours of particular sorting algorithms when input array size and array characteristics vary.

When the input consists of randomly generated integers, as we see in the plot-1 above, for the same sized array bubble sort algorithm has the worst running time and insertion sort has the second worst running time, which looks like a quadratic function. This result satisfies our theoretical expectation since both sorting algorithms have $O(n^2)$ complexity for average case. Even though their complexities are the same, as it is indicated in the result table above, bubble sort required more number of swap operations than insertion sort required. Therefore, bubble sort algorithm required more time to sort with respect to insertion algorithm. The other three algorithms, merge sort, quick sort and hybrid sort lies at the bottom of plot, which means they have the best running time and no significant difference can be observed among them in this experiment. When we looked at the plot they almost look like constant functions but this contradicts with our theoretical expectation, which is $O(n * logn)$ for average case. This situation might have resulted from [1]. But overall, the experimental results satisfies our theoretical expectation for the average case.

When the input consist of partially ascending integers, as we see in the plot-2 above, the behaviour of hybrid sort is dramatically different in our algorithms. Recall that the hybrid sort starts with quick sort and make recursive calls with subarrays which is divided with respect to pivot. After the array size becomes less than 20, we sort by using bubble sort. Therefore, in the worst case theoretically we expect $O(n^2)$ complexity since for larger arrays $log_2 n$ is also large with its multiplicity and these subarrays are randomly permuted among themselves. Therefore, bubble sort must execute for each such sub-random-arrays, which results in $O(n^2)$ complexity for average. It turns out that the experiment results are also convenient with the theoretical expectations since elapsed time quadraticly grows when size of array is increasing. For merge sort, due to partial order merge function makes less comparisons than random array as it can be seen in result table above. Our result and theoretical value $O(n * logn)$ might be called parallel except the reasons listed in [1]. For insertion sort, since most of the elements are grouped partially and we only need to control whether an element is in the right place in subarray which has the length $log_2 n$ and there are $n * log_2 n$ subarrays, we expect the elapsed time directly proportional to $O(n)$. Even though the graph of this algorithm looks like a constant function in the plot, the errors might have resulted from [1] again. For quick sort, since the array is not in descending order completely, partition function is able to balance the size of subarrays. Therefore, it is expected as $O(n * logn)$ since it is an average case, which is consistent with experimental results.

When the input consist of partially descending integers, for the same sized array elapsed time is the worst for bubble sort, $O(n^2)$ both theoretically and experimentally. The second worst algorithm is insertion sort since most of the array is not in order and we are manipulating it one by one, not dividing into smaller pieces. Therefore, it is not quite important to have groups of integers which are partially descending. Hence, it is $O(n^2)$ both theoretically in worst case and experimentally. The third one in this type of array is hybrid sort. The graph of Hybrid sort is also quadratic since we use bubble sort in base case and its worst case complexity is $O(n^2)$. Notice that since we are sorting in ascending order the move and comparison counts of these algorithms is more than partially ascending integers. Merge sort and quick sort is $O(n * logn)$ for such arrays and it can be seen in plot-3, there exist a small increase in these functions when array size is increasing. Since the partially ascending and descending arrays are constructed using mod operations on random algorithms in my implementation, this pseudo randomization might have resulted in increase in some particular patterns in some part of arrays which might also effect the experimental result.

[1] Possible causes of errors: Errors might resulted from the scale of the plot, sensitivity of time measurement, input distribution and limited inputs. If we had a scale which is more sensitive to smaller changes in elapsed time, the graph wouldn't be seem as constant but there would be increasing as we can see in result table. Also, one might take into consideration that comparison or move operations might take different times in different computers and also the other operations performed by the computer at the measurement time can damage the pure elapsed time measurement. Additionally, if we had examined much bigger sized arrays, the results would be more closer to the theoretical values. Furthermore, one can claim that errors might resulted from that it is not possible to completely random array and the array is constructed using a pseudo random algorithms.

begum.kara@dijkstra:~/CS202/HW1

Part 2-b-2: Performance analysis of partially ascending integers array

Array size: 1000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	0.044	3709	2709
Bubble Sort	0.104	5135	8954
Merge Sort	0.174	19951	5760
Quick Sort	1.003	5425	191924
Hybrid Sort	0.958	3357	191652

Array size: 5000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	0.235	22588	17587
Bubble Sort	0.687	37772	59921
Merge Sort	1.014	123615	35613
Quick Sort	18.501	30819	3824747
Hybrid Sort	18.3	16508	3821814

Array size: 10000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	0.479	47369	37369
Bubble Sort	1.491	82115	129908
Merge Sort	2.051	267231	76787
Quick Sort	68.375	63881	14275756
Hybrid Sort	68.009	32842	14268840

Array size: 20000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	1	100219	80219
Bubble Sort	3.324	180665	299879
Merge Sort	4.309	574463	164179
Quick Sort	256.968	131651	53793880
Hybrid Sort	256.006	65360	53778357

begum.kara@dijkstra:~/CS202/HW1

Part 2-b-3: Performance analysis of partially descending integers array

Array size: 1000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	3.383	499205	497947
Bubble Sort	7.379	1491623	499489
Merge Sort	0.168	19951	5629
Quick Sort	1.348	226171	140546
Hybrid Sort	1.332	223361	140018

Array size: 5000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	84.337	12492893	12486787
Bubble Sort	183.817	37448687	12497484
Merge Sort	0.96	123615	34553
Quick Sort	26.21	4663142	2761811
Hybrid Sort	26.054	4645685	2757592

Array size: 10000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	337.453	49982410	49970332
Bubble Sort	735.683	149887238	49994998
Merge Sort	2.04	267231	75108
Quick Sort	94.992	16952926	10029830
Hybrid Sort	94.547	16914958	10020144

Array size: 20000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	1350.97	199959943	199935915
Bubble Sort	2958.25	599759837	199989996
Merge Sort	4.28	574463	160974
Quick Sort	365.829	65324661	38652981
Hybrid Sort	365.191	65246008	38632900

[begum.kara@dijkstra HW1]$

begum.kara@dijkstra:~/CS202/HW1

Part 2-b-1: Performance analysis of random integers array

Array size: 1000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	1.76	257995	256989
Bubble Sort	5.803	767993	499121
Merge Sort	0.246	19951	8703
Quick Sort	0.201	20158	10364
Hybrid Sort	0.144	14155	8889

Array size: 5000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	42.117	6216281	6211279
Bubble Sort	164.712	18618851	12490478
Merge Sort	1.393	123615	55190
Quick Sort	1.201	106990	69651
Hybrid Sort	0.886	76033	61165

Array size: 10000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	168.094	24887144	24877133
Bubble Sort	717.251	74601440	49986484
Merge Sort	2.969	267231	120544
Quick Sort	2.643	260435	154416
Hybrid Sort	2.003	199073	136956

Array size: 20000	Elapsed Time(ms)	moveCount	compCount
Insertion Sort	678.179	100357983	100337977
Bubble Sort	3042.63	300953957	199988346
Merge Sort	6.315	574463	260971
Quick Sort	5.687	561811	343361
Hybrid Sort	4.432	437668	308699