Laboratory practice No. 2: Sorting algorithms

Sebastian Herrera Beleño
Universidad Eafit
Medellín, Colombia
Sherrerab1@eafit.edu.co

Jairo Miguel Marulanda Giraldo
Universidad Eafit
Medellín, Colombia
jmmarulang@eafit.edu.co

3) Practice for final project defense presentation

3.1

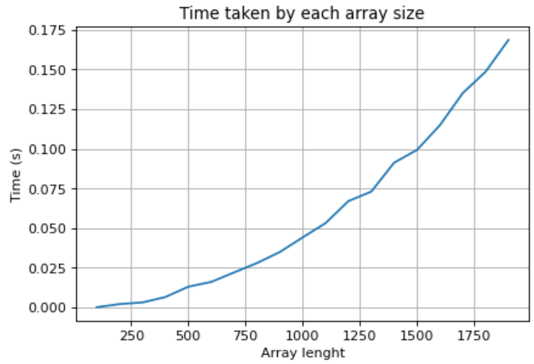
100	0.0
200	0.0010116100311279297
300	0.003961086273193359
400	0.008008718490600586
500	0.01101231575012207
600	0.016610145568847656
700	0.023960590362548828
800	0.02681589126586914
900	0.03654313087463379
1000	0.04280543327331543
1100	0.05031394958496094
1200	0.06364226341247559
1300	0.07583379745483398
1400	0.08860516548156738
1500	0.09787535667419434
1600	0.11363577842712402
1700	0.13083672523498535
1800	0.1474010944366455
1900	0.16482043266296387

Insertion Sort

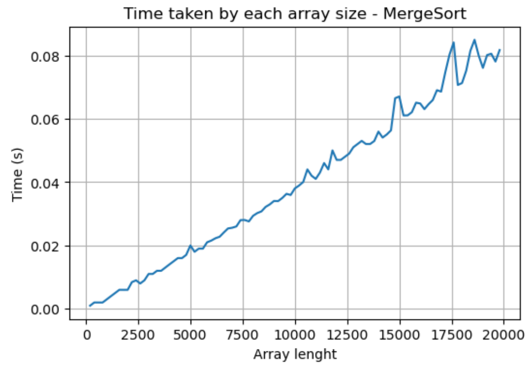
100	0.0009989738464355469
200	0.0
300	0.0
400	0.0009362697601318359
500	0.001991748809814453
600	0.002002239227294922
700	0.001997232437133789
800	0.003003835678100586
900	0.001990079879760742
1000	0.002999544143676758
1100	0.0028045177459716797
1200	0.004000663757324219
1300	0.003992557525634766
1400	0.003996133804321289
1500	0.0049877166748046875
1600	0.00499725341796875
1700	0.004996299743652344
1800	0.004999637603759766
1900	0.006999969482421875

Merge Sort

3.2



Insertion Sort



Merge Sort

3.3

This problem requires an algorithm that prioritizes time over memory, since images must be rendered quickly and efficiently, for otherwise the players experienced and gameplay may be damaged. Therefore, if we consider the enormous quantity of data that must be processed, and how quickly the time complexity of insertion sort increases, then we can conclude that other methods may give better results. Suh as merge sort, which has a much slower time complexity growth.

3.4

The logarithm appears in the merge sort complexity function. The why of this can be approached from several perspectives. The simplest, most immediate one is that the logarithm appears from the solution of the recursive equation used inside the algorithm.

Another, more analytical perspective goes as follows. Each time the function is called recursively, the length of the array, call it n, is divided by two, until the result equals one. This implies that the third recursive call of the method will divide n by 2^3, the fourth by 2^4, and so on. Let l be the total number of recursive levels needed to be called such that the length of the array received equals 1.

This implies:

n/(2^l) = 1
n = 2^l
l = log_2(n)

We can now estimate that it takes a complexity of n/2^a, with a being an integer, for each recursive call to end. Additionally, we know that a n/2^a its called on 2^a occasions, since each half of the array then goes on its own separate path until reaching length one. Then the total complexity of all calls for n/2^a arrays equals n. This applies for all array sizes.
Therefore, the total complexity of the algorithm is approximately the number of recursive levels required multiplied by n.
That is to say:

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

T(n) = l*n = n*log_2(n)

Note, this is an approximation made by analyzing the code, that ignores the exact number of operations made in every step yet helps to understand the formulation of the problem.

3.5

Since insertion sort complexity grows so quickly, we need the data set to be small. If this is not possible, we may also note that, unlike merge sort, insertion sort skips the positions that are already organized, so it may be faster if the data set is not too disorganized.

3.6

Let us now analyze in more depth the inner workings of the maxSpan algorithm.
We will say that the span of two equla numbers in an array is the number of elements between the two inclusive.
The maxSpan algorithm calculates the longest span inside a given array.
To do this it picks a position of the array to compare, it then iterates from the very last position of the array until it reaches the beginning of it. If any of the values evaluated in this iteration equals the chosen compared value, the distance between the two is compared with any other distance previously found and, if it happens to be larger, the value is saved, replaces the previous as the maximum distance and the iteration cycle is broken. The algorithm then picks another position to compare and repeats the process again, until it reaches the end of the array.
Finally, it returns the maximum distance found.

3.7

Let us now calculate the complexity of the suggested exercises.

3.7.1 exercise countEvens:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + n*c4 + c5, with ci constant.
Then:
T(n) = O(n)

3.7.2 exercise bigDiff:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + c2 + (n-1)*c3 + (n-1)*c4 + (n-1)*c5 + c8, with ci constant.
Then:
T(n) = O(n)

3.7.3 exercise enteredAverage:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

T(n) = c0 + c1 + c2 + c3 + (n-1)*c4 + (n-1)*c5 + (n-1)*c6 + (n-1)*c9 + c10, with ci constant.
Then:
T(n) = O(n)

3.7.4 exercise sum13:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + n*c4 + n*c5, with ci constant.
Then:
T(n) = O(n)

3.7.5 exercise sum13:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + n*c4 + n*c5 + n*c6 +c7 +c8, with ci constant.
Then:
T(n) = O(n)

3.7.6 exercise maxSpan:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + c2 + n*c3 + n*n*c4 + n*n*c8 + n*n*c9 + n*c10 + c11, with ci constant.
Then:
T(n) = O(n**2)

3.7.7 exercise fix34:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + n*n*c4 + n*n*c5 + c10, with ci constant.
Then:
T(n) = O(n**2)

3.7.8 exercise fix45:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + n*n*c4 + n*n*c5 + c10, with ci constant.
Then:
T(n) = O(n**2)

3.7.9 exercise canBalance:

Let T(n) be the complexity function of the algorithm with n being the length of the array.

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

We get for the worst case:
T(n) = c0 + c1 + n*c2 + n*c3 + c4 + c6 + c7 + n*c8 + c*c9 + c*c10 + c12, with ci constant.
Then:
T(n) = O(n)

3.7.10 exercise linearIn:

Let T(n) be the complexity function of the algorithm with n being the length of the outer array.
We get for the worst case:
T(n) = c0 + c1 + n*c2 + c3 + n*c4 + n*c5 + n*c6 + n*c11 + c12, with ci constant.
Then:
T(n) = O(n)
Note, we assume that either inner has the same length of outer or the element matches for inner are located in the very last positions of outer, such that the whole array outer must be evaluated.

3.7.11 insertion sort:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:
T(n) = (n-1)*c1 + (n-1)*c2 + ((n*(n-1))*2)*c4 + ((n*(n-1))*2)*c5 + ((n*(n-1))*2)*c6 + n*c7, with ci constant.
Then:
T(n) = O(n**2)

3.7.12 merge sort:

Let T(n) be the complexity function of the algorithm with n being the length of the array.
We get for the worst case:

T(n) = c1 + c2 + c3 + c4 + T(n/2) + c5 + T(n/2) + c6 + c7 + (n/2)*c8 + (n/2)*c9 + (n/2)*c10 + (n/2)*c11 + (n/2)*c12 + (n/2)*c14 + n*c15 + n*c16 + n*c17 + n*c18 + n*c19 + n*c20 + n*c21 + n*c22

with ci constant.

Which we can simplify as
T(n) = 2T(n/2) + O(n) = (c*n/2) + nlog_2(n), with c constant.
Then
T(n) = O(nlog_2(n)).
3.6
Let us analyze in more dept the inner workings of the maxSpan algorithm.

4) Practice for midterms

4.1 It would take approximately 10 seconds.
4.2 d
4.3 a

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

4.4 The time *complexity equals O(m*n). The matrix ocuppies aproximately 4*m*n bytes, so the space complexity equals O(m*n)*

4.5 a

4.6 b

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473