

DSA Exam

Exam Answers – Rez Graham

Mohamed hany

Amateur Student  Dubai, UAE

Data Structures & Algorithms Exam – Answer Sheet

1. Using Sigma Notation, we’ll do a summation by where n is max number of elements in the set A = {2, 4, 6, 8, 10, 12}.

N = 6.

I = 1.

X = A \* I.

X = ((1) \* (2)) + ((2) \* (4)) + ((3) \* (6)) + ((4) \* (8)) + ((5) \* (10)) + ((6) \* (12)).

X = 2 + 8 + 18 + 32 + 50 + 72.

X = 182

1. Using both Sigma & Product notation we’ll do another series. Set B is {3, 6, 9, 12}.

X = Summation of Set B + Root of Products of Set A.

Summation of Set B = ((1) \* (3)) + ((2) \* (6) + ((3) \* (9)) + ((4) \* (12))

Summation of Set B = 3 + 12 + 27 + 48.

Summation of Set B = 90.

Products of Set A = (2) \* (8) \* (18) \* (32) \* (50) \* (72).

Root of Products of Set A = 5760.

X = 90 + 5760.

X = 5850.

1. Like question 1 except the upper limit is halved. So

X = 2 + 8 + 18

X = 28.

1. C is the intersect of sets A & B. D is the union of both sets.

C = {6,12}

D = {2,3,4,6,6,8,9,10,12,12}

1. The order from fastest to slowest is the following in terms of growth rate (I looked up some refreshers here. I took his in uni but forgot how to do it, so a fresh explanation would be needed):

2^N – Exponential time growth.

N^2 – Quadratic time growth.

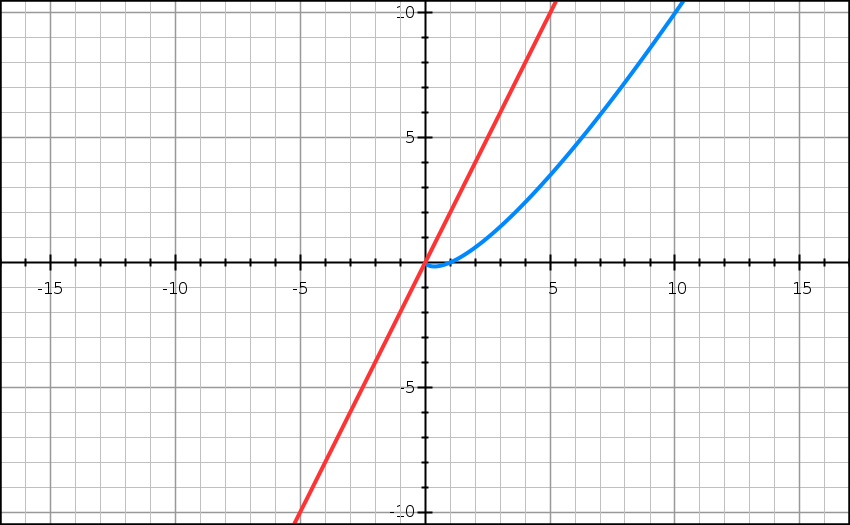
N \* Log(N)

N – Linear Time Growth.

Log(N) – logarithmic time growth.

1 – Constant Time Growth.

1. F(X) has a growth rate of O(2N). so, it grows at twice the constant time speed, so it’s growth would be faster than O (N \* LOG(N)). Therefore H(X) is faster in execution time as its growth is slower.



Red: Plotting of 2N

Blue: Plotting of N \* LOG(N)

1. Below are the descriptions for each notation:

**O – Big O Notation:** It’s often used for calculating the upper bound or worst-case scenario for the growth rate of an algorithm.

**Ω - Big Omega Notation:** is used to describe the best case or lower bound for the growth rate of an algorithm.

**Ɵ – Big Theta Notation:** Is used to describe the exact boundaries so for constant x “growth rate is no less than Y and no greater than Z” or as they academically call it a “tight bound” or region in linear programming.

**o – Little O Notation:** like Big O but the key difference here is that the upper bound can never be reached.

**w – Little Omega Notation:**  I’m guessing it’s like little O notation in the sense that the lower bound is defined but cannot be reached as well.

1. For the function with heavy up front constant time cost. I believe I’d choose it on the following grounds:
   1. The algorithm or number of executions can justify the performance benefit of logarithmic time cost. So, for example, if the function or algorithm is called 500 times in my system both algorithms would be terrible choices.

If K is at least < ½ N. I’d choose the function with upfront cost as it’s execution time would flatten out near the constant K. so overall, it’ll always be less then than N. but if K > N or equal N. then at the beginning the constant time function would be a better choice but as the input set grows they’ll be evenly matched.

I’d say it all depends on context & the number of executions for the algorithm so one is not specifically better.

1. O (N \* LOG(N)) where N is the size of the data set. This the average / most common lower bound (best case) for most comparison sorting algorithms.
2. given that random sort does not require comparisons and just does random swapping. Then It would perform at O (1) in the best case as it does not require any nest iterations and the set is sorted after 1 run.

As for the worst case, I’d say it depends on what happens when random sort finds out that the set is not sorted. However, the worst case would be O(N) since it will always have to perform a linear search through the set regardless if the set ended up being sorted or not.



|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Best Case** | **Worst Case** |
| LINEAR\_SEARCH | O (1) | O(N) |
| QUICK\_SORT | O (N \* LOG(N)) | O(N^2) |
| BINARY\_SEARCH | O (1) | O(LOG(N)) |

Complexity of FUNCTION-1: O (1) or || O(N) as it does a linear search.

Complexity of FUNCTION-2: Best Case of (QUICK\_SORT + BINARY\_SEARCH) || Worst Case of (QUICK\_SORT + BINARY\_SEARCH)

So, for FUNCTION-2:

Best case is: (N \* LOG(N)) + (1) = N \* LOG(N) (ignoring constants).

Worst case is: (N \* LOG(N)) + N^2.

If we take N as common factor, N \* (LOG(N) + N)

FUNCTION-1 Performs faster in both cases.

1. For string comparison, the there is no possible way to achieve less than O(N) since there you’ll most likely loop through the strings for the comparison. so, it’s O(N) in the worst case and O (1) in the best case if the first two characters of the strings match. Where N is the size of the shorter string. This is on the assumption that both strings do not require sorting.