

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:

1 – Constant Time Growth.

Log(N) – logarithmic time growth.

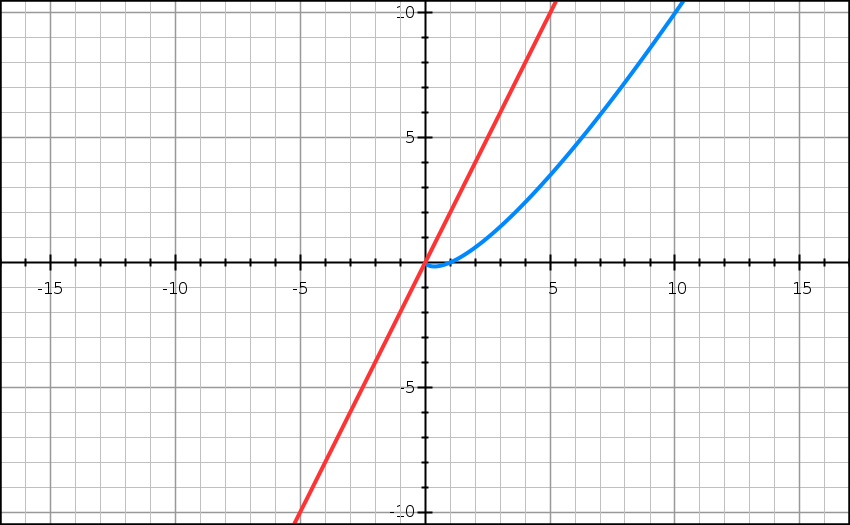
N – Linear Time Growth.

N \* Log(N)

N^2 – Quadratic time growth.

2^N – Exponential 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. I’m not sure if in this case you should ignore the constants on N because it would make a big difference.



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.
  2. 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.
2. For this question I spent at least 2 hours looking at all the possible solutions. My conclusions are the following: it’s impossible to do this in less than O(N) time since u need O(N) to traverse all the array once.

Furthermore, if it’s an unsorted array it’s not possible to do in less than O(N) time. I found a hacky trick that changes bits or values in the array to do the whole thing in O(N) time but that is a bad idea.

Therefore, the best I can come up with is to sort the array then do a linear search. The complexity is the following:

Best Case: (SORTING + LINEAR SEARCH FOR CHECK) = O (N \* LOG(N)) + O (1) = O (N\* LOG(N)).

Worst Case: (SORTING + LINEAR SEARCH FOR CHECK) = O (N^2 + N).

Now I couldn’t figure out how to use std::sort with a \* array. So, I ended using the naïve approach which has a best case and worse case of N^2. Both approaches have O (1) memory usage but the second approach is not the fastest.

bool CheckForDuplicates(int\* pArray, int size)

{

for (int i = 0; i < size; ++i)

{

for (int j = 0; j < size; ++j)

{

if (j != i && pArray[i] == pArray[j])

{

return true;

}

}

}

return false;

}

1. For this question, I went ahead with the naïve implementation. I didn’t know any specific data structure or algorithm to solve this in a better way. Although I did identify the pattern. There is a good chance I totally misunderstood this one.

IF A DO 1 (Print Fizz)

IF B DO 2 (Print Buzz)

IF A & B DO 3 (Print Fizz buzz)

IF NONE DO 4 (Print number)

void FizzBuzz(int maxValue)

{

for(int i = 0; i < maxValue; ++i)

{

std::cout << GetPrintMessage(i);

}

}

std::string GetPrintMessage(int number)

{

if(IsMultiple(number, 3))

{

return "Fizz";

}

if (IsMultiple(number, 4))

{

return "Buzz";

}

if (IsMultiple(number, 4) && IsMultiple(number, 3))

{

return "FizzBuzz";

}

return "Number is: " + number;

}

bool IsMultiple(int number, int factor)

{

return number % factor == 0;

}

1. For this question I wrote a simple binary search since it works well on sorted data structures (Refreshed on it before answering).

int FindIndex(const int\* pArray, int size, int valueToFind)

{

assert(size > 2);

assert(pArray != nullptr);

int start = 0;

int end = size - 1;

for(int i = 0; i < size; ++i)

{

int mid = start + (end - start) / 2;

if (end < start)

{

return end;

}

if(pArray[mid] == valueToFind)

{

return mid;

}

if(pArray[mid] > valueToFind)

{

end = mid - 1;

}

else

{

start = mid + 1;

}

}

}

The key take-away from the code is the check for end < start. This means that the value is not in the array if it isn’t I return the smaller of the two indices. The worst case for a binary search is O (LOG(N)) therefore for this algorithm the worst case is O (LOG(N)). A linear search could have just been fine here and more straight forward as well. But I assumed u stated sorted as a hint here.

1. Here’s my educated guesses for these, I assume index is the value of the subscript operator in the case of arrays and being the key in the case of hash tables.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Search | Index | Insert | Delete | Delete Me |
| Unsorted Array | O (N) | O (N) | O (1) | O (1) | O (N) |
| Sorted Array | O (LOG(N)) | O (N) | O (N) | O (N) | O (N) |
| Linked List | Don’t Know | Don’t Know | Don’t Know | Don’t Know | Don’t Know |
| Hash | O (1) | O (1) | O (1) | O (1) | O (N) |

1. For std::vector, it grows in size every time a push\_back() is called when the default size of the container is completely full. So, for insertions that do not trigger this resizing, the insertions would always be at index (vector size – 1) so it’s O (1) constant time. However, for insertions that resize the vector, it would have to resize & copy all elements up to the current index in the new array and then finally place the new element at index (size before re-sizing). This would take O (N) time where N being the number of elements present before resizing within the container.

As for worst case Memory complexity. Each time the vector resizes I’m not exactly sure if it’s current capacity doubles or does it just simply resize to be (current size + 1). Regardless it will boil time to O (N) where N being the number of elements previously present in the array as they all are copied/re-allocated into the new resized array.

1. I only know that a hash table is the basis for a dictionary or a map and can relatively explain how the indexing would work but I def can’t write up the formal definition as I never took it my uni.
2. I guess it’s either when the same value is used with 2 different keys or “hashes” so that during lookup the same hash or key has 2 corresponding values.
3. So, we have some main properties for the data we’ll be using:
   1. It’s a large data set (1000s)
   2. It’s relational (each name has commonality indicator related to it, probably referenced often to pick a name).
   3. 3 distinct data sets (Male First name, Female second name, Surnames)
   4. Must be loaded on start into the data structure. So read/write speed might be relevant.

As for operations, we would need the following based on what I could perceive.

* + - ***Find*** *names via value or (aka names used less than K amount of times or range (min - max)*
    - ***Find range*** *get a range of names, this can be used if the random name generator does layered randomization (fetch random set -> shuffle it -> get random name…etc.)*

Now the look up / find operation must be quite fast on a large data set. Which also seems to be sorted in order of commonality. I’m not sure if multiple names might have the same commonality or is each name has a unique value. Regardless here’s how I’d approach the problem in both cases.

I’d use a sorted dictionary / hash table for each data set. The look up seed is O (1) and it’s relatively the best choice for large data sets. I’d load the data from a CSV File into the hash table on start.

1. For the data structure to be used. It needs the following properties:

* Fast Look up (Lots of checks for interactions).
* Mutable (Changing relationship variables).
* Insertion.
* Iteration.

For the next of questions I took a look at a number of data structures but most seemed too unrealistic so I chose to default to an std::vector. My reasoning was with the current architecture it’s difficult to map each relationship to it’s respective Game object since the relationship struct contains both pointers.

Furthermore, taking a peak on further questions it seems like iterating over the relationships for consistent checks will be required. Which is not as friendly with data structures such as maps, sets, queues…etc. that might come to mind. This might not be the right choice at all but coming from a practical stand point I’d start simple and see how things go.   
Lastly, with std::vector sorting the data can be an easy task to achieve which can even lead to using binary search as an efficient searching algorithm for finding specific things.

1. The worst-case scenario would be O (N) where N being the number of other romantic relationships. This is because we’ll be looping over the std::vector of relationships and checking if the current relationship is a romantic one while also making sure it’s not the relationship we just changed to romantic. This is my first guess at it.
2. I’m not sure if you mean express interest initially (changing type to romantic) or including the addition in Q.22. the reason I’m doubting is that you didn’t hint anything to the previous question. Regardless here’s my fair go at it.

We initially need to do 2 things with 3 constraints:

1. Set the relationship type to romantic if express interest is successful.
2. Set all existing romantic relationships (if more than one) to friends.

*Constraint:* memory allocations O (1), in the same amount of time O (1), only architecture changes. By architecture here I’m assuming I’m not allowed to change game design.

Our current code flow is the following in a strictly sequential sense:

While in (Conversation):

If (Friend) -> Try Express Interest.

If (Expressed Interest) -> Change to Romantic & If (Romantic > 1) Reset Others to Friends.

End (Conversation)

I’m also assuming while in the conversation you’d have a way to reference the game object (student) you’re currently conversing with.

Here’s my go at it using all the previous assumptions:

public void RelationshipLoop(Relationship other)

{

while (IsInConversation())

{

// some other stuff here.

if (other.Type != RelationshipType.Friends) continue;

if (TryExpressInterest())

{

other.Type = RelationshipType.Romantic;

}

}

}

Now I don’t strictly know what I can change to make it run at O (1) time. for O (1) memory this is strictly easy since I’ll be setting variables to new values. But nevertheless, here’s how I’d look at it:

Somehow, I need to be able to easily find the relationships of type romantic from the std::vector or which ever data structure makes the relationship types O (1) look up and set each to type friends. Now I know I still didn’t answer the question but for the sake of the mentorship I think this will help you more find out where exactly I need improvement.

1. The worst-case performance on my guess would be also O (N) since we’ll have to iterate over our data structure & add any relationship with value 75 and above.

I’d say if there is a way to make look up based on certain criteria (relationship value) O (1) then we can make it faster, but now I don’t know how I’d architect that or if there is a data structure that can simplify the problem. I assume because it’s been a long time since I’ve applied these concepts to daily stuff.

Although at second look perhaps a sorted tree (binary or some sort of tree type) can help “bucketing” relationships based on these values. Therefore, the look up would be trivial by just returning the sub tree of the said bucket. I haven’t applied trees before, so I don’t know how I’d go about it, but I think that would be my last resort to solve the problem.

However, if the tree was sorted based on relationship value ranges (0 – 50) …etc. && type it should be a relatively quick breadth first search that just returns the branch with values >= 75.

1. Using my previous answer (std::vector) yes that would make the sorting based on criteria required. I’d try to split the vector in 3 buckets or sub vectors. Each vector is based on these thresholds so per vector the worst case would be for sorting which will average on O (N \* LOG(N)).

However, if I was using a tree structure, that means I’d sort each main branch on these buckets. So, branch A -> >= 50 && romantic partners. B-> >= 75 & friends. C -> > 0 && Family.

I think the average for sorting a tree is O(LOG(N)) and worst case would be O (N) where N is the number of nodes in the tree.