Data Structures

A level-order traversal in a binary tree requires which data structure

Top of Form

subset sum problem

number of swapping operations in bubble sort <https://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html>

How to find the complexity of following task:-

int fun(int n)

{

for (int i = 1; i <= n; i++)

{

for (int j = 1; j < n; j += i)

{

// Some O(1) task

}

}

}

What is the data structure implemented in quicksort

Set, stack, list, depth first search

How are variables and memory managed in Python? Does it have a stack and a heap and what algorithm is used to manage memory?

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching. The algorithm used for garbage collecting is called **Reference counting**. That is the Python VM keeps an internal journal of how many references refer to an object, and automatically garbage collects it when there are no more references refering to it.

Bottom of Form

Insertion sort

N^2 worst case complexity

Although it is one of the elementary sorting algorithms with O(n2) worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

**Why Numpy instead of python lists?**

NumPy's arrays are more compact than Python lists -- a list of lists as you describe, in Python, would take at least 20 MB or so, while a NumPy 3D array with single-precision floats in the cells would fit in 4 MB. Access in reading and writing items is also faster with NumPy.

a Python list is an array of pointers to Python objects, at least 4 bytes per pointer plus 16 bytes for even the smallest Python object (4 for type pointer, 4 for reference count, 4 for value -- and the memory allocators rounds up to 16). A NumPy array is an array of uniform values -- single-precision numbers takes 4 bytes each, double-precision ones, 8 bytes. Less flexible, but you pay substantially for the flexibility of standard Python lists! NumPy is not just more efficient; it is also more convenient. You get a lot of vector and matrix operations for free, which sometimes allow one to avoid unnecessary work. And they are also efficiently implemented.

First, numpy stores array elements using fixed physical record sizes. So, record objects need to all be the same physical size. For this reason, you need to tell numpy the size of the string or save a pointer to a string stored somewhere else

Merge sort

It produces a sorted output array in at max 6nlog2n + 6n operations

At each level it has 2**j** problems to solve and each problem has n/2**j** recursive calls to solve where j is the level number with 0 as the first level and n is input array size to be sorted.

Merge operations merge(output array, sorted array a, sorted array b)

Guiding principles on design of algorithms

1. Worst case analysis for upper bound
2. average case analysis under different frequencies, benchmark inputs based on domain knowledge
3. won’t pay much attention to const. factors, lower order factors as these are more dependent on architecture/programmer and have very less predictive power
4. asymptotic analysis i.e. to be more focused on large input sizes

Insertion sort –

Complexity is ½ n2

Principle of counting inversions-

def mergeSort(alist):

global cnt

cnt = 0

if len(alist)>1:

mid = len(alist)//2

lefthalf = alist[:mid]

righthalf = alist[mid:]

cnt+= mergeSort(lefthalf)

cnt+= mergeSort(righthalf)

i=0

j=0

k=0

while i < len(lefthalf) and j < len(righthalf):

if lefthalf[i] <= righthalf[j]:

alist[k]=lefthalf[i]

i=i+1

else:

alist[k]=righthalf[j]

j=j+1

cnt = cnt + len(lefthalf) - i

k=k+1

while i < len(lefthalf):

alist[k]=lefthalf[i]

i=i+1

k=k+1

while j < len(righthalf):

alist[k]=righthalf[j]

j=j+1

k=k+1

return cnt

def count\_inversions(a):

return mergeSort(a)

Divide, conquer and combine

For eg a 6 element array –

Largest number of inversions can be n(n-1)/2 i.e. 15

Graphs

Vertices (V), node

Edges (E), links can be Directed/undirected