Behaviour Questions

Why do you want to work at Google? - Google is a successful company. - Learn what makes these companies successful (from a technical perspective) and how they work. -If you don't see early on what structure does look like - it will be hard to set goals later on. - Technically challenging, and I think it would be nice to understand how engineering teams work in a company as large as google.

Tell me about yourself

I like coding and thinking logically. I actually came from a research orientated background. I like both the idea of pushing the boundaries of what I know, and I also like making things.

I started to realise I like forming actual real ideas into code, more than I like the research ideas.

In my spare time, I read books, exercise.

Iterator for tree

```
def iterator(node):
    if node is None:
        return
    yield from iterator(node.left)
    yield node.val
    yield from iterator(node.right)
```

Prefix sum

```
P = [0]
for x in A:
    P.append(P[-1] + x)
```

Binary search (Duplicates)

```
    low = 0
    high = len(nums)-1
    index = -1
    while low <= high:
```

```
            mid = (high + low) // 2
            if nums[mid] == target and (mid == 0 or nums[mid-1] != target):
                    index = mid
                    break
            elif target > nums[mid]:
                    low = mid + 1
            else:
                    high = mid - 1
```

#Insert left bound binary search/right bound binary search

Merge sort

Requires extra memory Stable

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        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
            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
    print("Merging ",alist)
```

Quick sort

In-place Not stable

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

       splitpoint = partition(alist,first,last)

       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)


def partition(alist,first,last):
   p = alist[first]

   m = first
   for k in range(first+1, last+1):
      if (alist[k] < p):
          m += 1
          swap(a,k,m)
   swap(a,i,m)
   return m
```

Heapify

```
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
         tmp = self.heapList[i // 2]
         self.heapList[i // 2] = self.heapList[i]
         self.heapList[i] = tmp
      i = i // 2


def percDown(self,i):
  while (i * 2) <= self.currentSize:
```

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	O(n)	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2 (improved with flag)	$O(n^2)$	O(n)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	No	Yes
Radix Sort (non-comparison based)	**O(n)**	O(n)	No	yes
Quick Sort	$O(n^2)$	O(n log n)	Yes	No

Figure 1: alt text

```
    mc = self.minChild(i)
    if self.heapList[i] > self.heapList[mc]:
        tmp = self.heapList[i]
        self.heapList[i] = self.heapList[mc]
        self.heapList[mc] = tmp
    i = mc
```

Successor

```
def inOrderSuccessor(root, n):

    # Step 1 of the above algorithm
    if n.right is not None:
        return minValue(n.right)

    # Step 2 of the above algorithm
    p = n.parent
    while( p is not None):
        if n != p.right :
            break
        n = p
```

```
        p = p.parent
    return p
```

Bits

```
def getBit(num, i):
    return ((num&1<<i))

def setBit(num,i):
    return num | i << i
def clearBit(num,i):
    mask = ~(i<<i)
    return num & mask
```

Permutation / Combination

```
from itertools import permutations, combination
# Permutation generates n!, combination does nCr

def all_perms(elements):
    if len(elements) <=1:
        yield elements
    else:
        for perm in all_perms(elements[1:]):
            for i in range(len(elements)):
                # nb elements[0:1] works in both string and list contexts
                yield perm[:i] + elements[0:1] + perm[i:]


def combinations_by_subset(seq, r):
    if r:
        for i in xrange(r - 1, len(seq)):
            for cl in (list(c) for c in combinations_by_subset(seq[:i], r - 1)):
                cl.append(seq[i])
                yield tuple(cl)
    else:
        yield tuple()

        val set = {"A", "B", "C"}

        val sets = {}
```

```
            val set = {"A", "B", "C"}

            val sets = {}


result.add({})
for item in set:
    for set in result:
        result.add(set + item)
```

NP-P Problems

1. Vertex Cover
2. 3 SAT
3. Independent Set
4. Hamiltonian Path Decision Problem
5. 0/1 Knapsack

UFDS

x = FindSet(i)

y = FindSet(j)

if (x != y) // !IsSameSet(i, j)

```
if rank[x] > rank[y]

      p[y] = x;

        else p[x] = y;
```