Synthesis 1

Q1.1 d

Q1.2 $f(n) = O(n^3)$

Q1.3 $K = 0$

Proof: $T(1) = 1$

$$T(n) = 32T\left(\frac{n}{2}\right) + n^k$$

from master theorem $a = 32$, $b = 2$

if $T(n) = \Theta(n^5 \log n)$ that mean $K \geq 0$

$\Rightarrow n^{\log_2 32} \log^k n$ $\Rightarrow$ case 3

$\Rightarrow$ $\boxed{K = 1}$

Q1.4 11 swaps

Q1.5 76

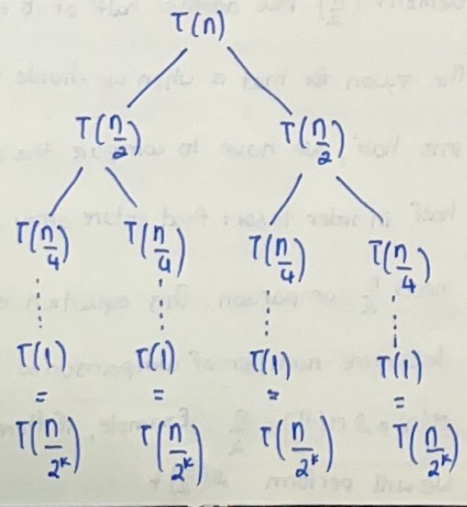
Q2.1 see another file (Q2.1.png)

Q2.2 see another file (Q2.2.py)

Q2.3 Determine a recurrence relationship of $T(n)$

Explain why the recurrence relationship is true

$T(n)$ is the maximum number of guess and it keeps splitting in half, hence:



from here we can see the recurrence relation is

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

~~at when~~ adding it all up, we have

$$T(n) = T\left(\frac{n}{2^k}\right) + k \text{ time}$$

$\frac{n}{2^k} = 1$ (1 is the base case)

$\Rightarrow 2^k = n$

$\log_2 2^k = \log n$

$K = \log n$

$\Rightarrow \Theta = \log n$

proof with master's theorem:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

from master's theorem $\Rightarrow a = 1$, $b = 2$

and $f(n) = \Theta(n^{\log_2 1} \log n)$ $\left\{ \begin{array}{l} n^{\log_2 1} = n^0 = 1 \\ \log n \text{ need to be equal to 1 too} \end{array} \right.$

$\Rightarrow$ second case of master's theorem $K \geq 0$

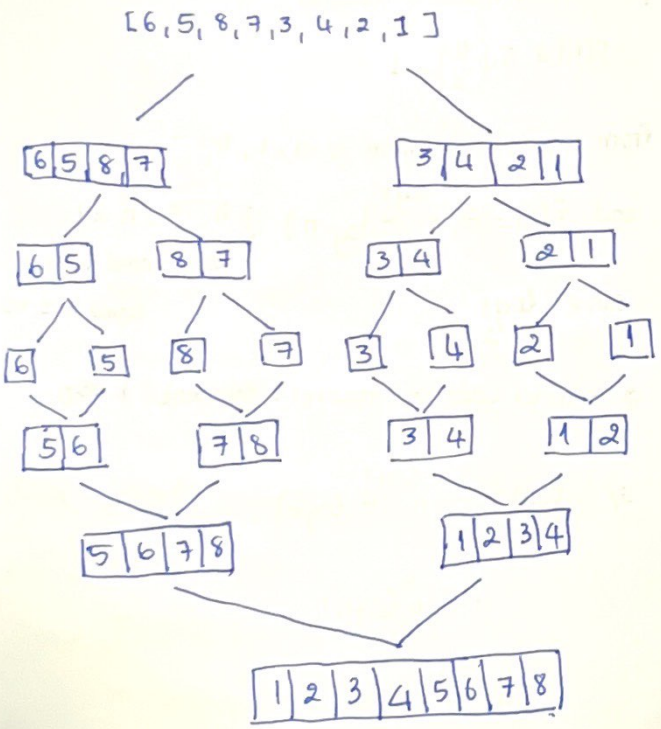
$\Rightarrow T(n) = \Theta(n^{\log_2 1} \log n)$

$= \Theta(n^0 \log n)$

$$\boxed{T(n) = \Theta(\log n)}$$

Q2.4 I will not play the game if you give me $ 15 and each game I would lose $ 1. The reason is the maximum or worst case for me to guess the correct word is $\log_2 267\,751 \approx 18$ time. This would means, there's chance that I could spend $ 18 to guess the word right. Since you only give me $ 15, I'll lose $ 3. So no I will not play the game.

Q3.1 Perform merge sort algorithm



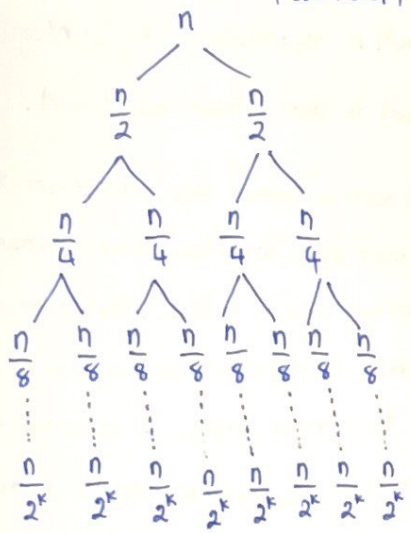
- 1st comparison is when we are trying to merge (5,6)
- 2nd comparison is (8,7)
- 3rd comparison is (3,4)
- 4th comparison is (1,2)
- 5th comparison is (5,7)
- 6th comparison is (6,7)
- 7th comparison is (1,3)
- 8th comparison is (2,3)
- 9th comparison is (1,5)
- 10th comparison is (2,5)
- 11th comparison is (3,5)
- 12th comparison is (4,5)

Q3.2 Clearly explain why $M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$ if n is an even number

Since n is an even number, when we divide n into 2 half, we will get $\frac{n}{2}$ (imagine we are trying to sort it using merge sort so we have to use divide and conquer). Hence, we could write this equation as $M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$. What the equation is saying is in order to sort an array a with n element, the number of comparison needed would be twice the number of comparison needed to sort A with half of its element $\left(\frac{n}{2}\right)$ plus another half of its element $\left(\frac{n}{2}\right)$. The reason for that is when we divide the array into half, we have to compare the array in each half in order to sort that entire array, thus we need $\frac{n}{2}$ comparison. This equation is used to determine number of comparison so it would be $M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$.

Q3.3 if n is power of 2, prove that $M(n) = \frac{n}{2}\log n$

$M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$ { n is splitting into $\frac{n}{2}$ two times; each step, take $\frac{n}{2}$ time



Total are K steps and we assume in that last step of dividing $\frac{n}{2^k} = 1$ (base case $T\left(\frac{n}{2^k}\right) = 1$)

$\Rightarrow \frac{n}{2^k} = 1 \Rightarrow 2^k = n$

$\log_2 2^k = \log_2 n$

$k = \log n$

since, each step take $\frac{n}{2}$ time

$\Rightarrow \boxed{\Theta\left(\frac{n}{2}\log n\right)}$ or $\Theta(n\log n)$

Q3.4 Let A be a random permutation of [1,2,3,4,5,6,7,8]. Determine the probability that exactly 12 comparison are required

in merge sort, we need to divide the array into half, each half will then have to divide again and again until we get to 1 elements in the array, then we will merge it back by doing the comparison

- since we have 8 elements in total → total = 8
- probability for a single comparison = $\frac{1}{2}$ (divide into half)
- we want to get exactly 12 comparison

⇒ binomial distribution:

$P = 12C_8\left(\frac{1}{2}\right)^8\left(\frac{1}{2}\right)^4$

$= 12C_8\left(\frac{1}{2^{12}}\right)$

$= \left(\frac{12!}{8!4!}\right)\left(\frac{1}{2}\right)^{12}$

$= \left(\frac{9\times\cancel{8}\times\cancel{7}10\times11\times12}{1\times2\times3\times4}\right)\times\frac{1}{4096}$

$= \frac{495}{4096}$

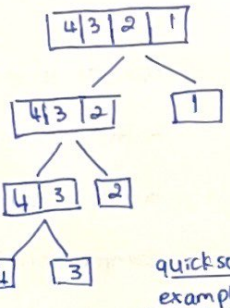
$\boxed{P = 0.1208 = 12.08\%}$

Q4.1 Insertion sort would be faster than selection sort in our case because the array we are trying to sort is already sorted (each element is the same). Since it is already sorted, when we run the insertion sort, the code does go to the first for loop to check each element, but the when it goes into while loop to do the comparison to see if it is greater than the previous element, it doesn't execute anything underneat it since it is already sorted. Hence, this could potentially reduce the time complexity from $O(n^2)$ (1 for loop and 1 while loop) to $O(n)$ (1 for loop only since while loop never get executed). For selection sort, we first have to run through for loop in range len of the array, the we have to run through another for loop starting at

position i+1 to length of the array. In that second for loop, we check the condition to see if array[i] is greater than array[i+1] then do the swap latter if need be. Due to this reason, even if in a good scenerio where the element in the array is already sorted, it the code still execute 2 for loops. Hence, the time complexity is $O(n^2)$.

∴ Insertion sort is better than selection sort in the best case scenerio ($O(n) < O(n^2)$)

Q4.2 Heapsort will be faster than quick sort in the case of having the list order in a descending order and the right most element (smallest number) is the pivot. This is because for quicksort this is its worst case. Worst case happens if we pick the smallest or largest element and the array is in sorted or reversed sorded order. The time complexity would be $O(n^2)$ because in each time the comparison would be n, then (n-1) then (n-2) all the way to 1. Hence $\Theta\left(\frac{n(n+1)}{2}\right) \approx O(n^2)$



quicksort example

In heap sort, the time complexity regardless of best or worst case scenerio is nlogn. The reason is in order to perform heap sort, first we will have to do insertion to the heap then deleting from heap. Insertion bee takes nlogn time cause it takes logn time to insert n element (heap is a binary tree, that's why it is log n). Deletion also takes nlogn time cause it has to delete n element at logn time at it deletes min element first. Hence in total it take 2nlogn time which is equivalent to O(nlogn)

∴ Heapsort is better than quick sort.

Q4.3 Bucket sort is better than bubblesort. Because in bubble sort in its worst case and average case, the time complexity is $O(n^2)$. That is because has to compare its adjacent number one by one till it gets the sorted array. In bucketsort, the average case is $O(n+k)$ where k is the number of bucket and n is number of element and in the worst case, $O(n^2)$. It is in the worst case if we have all element in the same bucket; hence, the linked list in that bucket will have to traversed every time the element is added. This make the time complexity to be the sum of all the number from 1 to n making $O(n^2)$. However, we could create an algorithm to avoid having every elements in one bucket. This makes the time complexity to be linear.

∴ We can conclude that bucket sort is better than bubble sort.

Q4.4 since we are using an array A from a set of digit 0 to 9, in this case counting sort will be better than merge sort. The reason is in counting sort, the time complexity is $O(n+k)$. That is because in the algorithm, we will loop through each element then incrementing the counter for each element. Then we loop through the count to get add previous value to get summation of count. At the end, we loop through the element in reverse so that we could use the item key to index the count array, decrement that count array value and use that decremented value as an array index to copy item to a sorted array.

Since we initiate k counter and loop through n element in the unsorted array, that take a linear time.

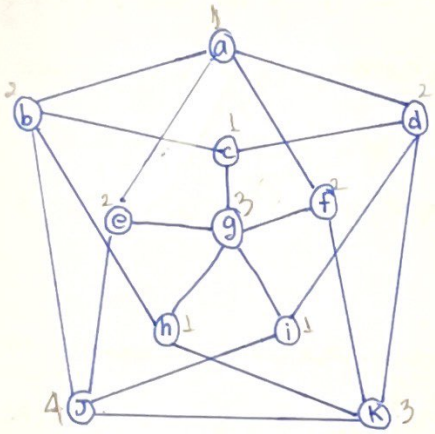
Hence it takes $O(n+k)$ time.

As for mergesort, it is a divide and conquer sorting algorithm. In merge sort, we have to devide n elements in an array to $\frac{n}{2}$ until we get to the base case where we only have 1 element in the array. Since we keep splitting it in half, the runtime is $\log n$. Then after that we have to merge all n elements to get and perform swap; hence, it take n time.

In total it takes $n \log n$ time.

∴ counting sort is better than merge sort if the array has n elements that is chosen from number 0 to 9. That's because counting sort has $O(n+k)$ and merge sort has $O(n \log n)$

QS.1 Let G be the graph below, with 11 vertices and 20 edges. Clearly explain why $\chi(G) = 4$



all vertices are:

a → b, e, f, d
b → a, c, h, J
c → b, g, d
d → a, c, i, K
e → a, g, J
f → a, g, K
g → c, e, f, h, i
h → b, g, K
i → J, g, d
J → b, e, i, K
K → J, h, f, d

- we will sort the node by ascending order, we will start by node a. From node a, we will mark it as color 1, the nodes adjacent to a can't have color 1 so we will mark it as color 2.

Now, we will look at node b, b has color 2, so any nodes adjacent to b can't have color 2. so b → a has color 1, correct. b to C ~~is 2, we will~~ doesn't hav any color and we know now we have use 2 colour 1 and 2, so we will reuse color 1 in C.

h and J doesn't have any colo now and we know it can't be 2 so we will mark it 1.

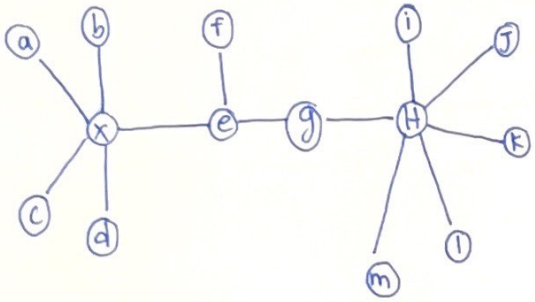
- Node C has neighbor's node b and d. C → b and C → d is 2 and not 1, so that's correct. C is also neighbor with G. C is 1 that mean G can't be 1, so we will mark it as 2.
- from node d, it neighbors a and a and d are not the same, so that's correct. Same idea for ~~me~~ another neighbor of d, C. i and K do not have any color and we know it can't be 2, so we will mark it as 1.
- Node f neighbor with a, g and K. From f to a and K is correct cause node f to g and K have different color. But from f to g, it is the same color 2 and 2. Hence, we need to change node g to 3 cause it can't be 1, (we would have use 1 and not 2 in the beginning if it is applicable).
- Node G, neigbors of node G have all different color from 3, so it's correct.
- Node h neighbors with b, g, k. b and G are correct cause its color is different from node h. However, K is not. If we change K to color 2, it will not work cause it will be the same as d. we could change that to 3.
- Node i neighbor with J, g, d. g and d are correct. we need to change J. If we change J to 2, ~~we~~ it won't work cause it would violate b. Change to 3, ~~we~~ not work cause it violate K. Hence, we will change it to 4.
- J neighbor with b, e, i, K and they are all have different color from J.
- K neighbor with J, h, f, d and they all have different color.

Q5.2 see another file (Q5.2.py & Q5.3.py)

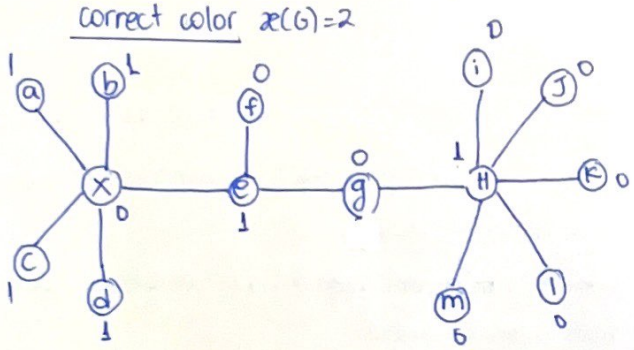
Q5.3 see another file (Q5.2 & Q5.3.py) & (Q5.3.png)

Q5.4 see another file (Q5.2 & Q5.3.py)

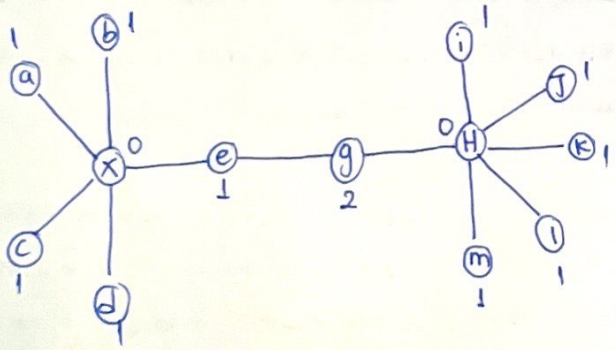
Visual graph



correct color $\chi(G)=2$



my algorithm suggests $\chi(G)=3$



number 1, 2, 0 represent color like color 0, color 1, color 2.

Q5.5 If an arbitrary graph has chromatic number of 2, that means the graph is a bipartite graph. We would assume that our graph has even number of nodes if it is a cycle graph (chromic number will not be 2 if it is a cycle graph with odd number of node).

With the description above, we could use an algorithm to check if the graph if bipartite cause if it is, we will return true ($\chi(G)=2$) else return false ($\chi(G)!=2$). We will use breadth first search for that. Algorithm:

1. assigne U color to the source vertex
2. mark the source vertex's neighbor to V color
3. mark the neighbor's ~~color~~ neighbor color to U
4. when assigning, check if there is a neighbor that has the same color as current vertex cause if there is then return False cause the graph is not bipartite anymore. If the operation is successful and ~~each~~ nodes are seperated in color U and V, then return True.

The complexity is $O(n^3)$ where n is the number of node or vertices. This run time is $O(n^3)$ because it would be $O((V+E)V)$ where V is ~~node~~ vertices or vertex and E is edge. Since n is number of nodes edges $\Rightarrow n^2$ is number of edge.

$\Rightarrow O((n+n^2)n) \Rightarrow \boxed{O(n^3)}$