1. $3n^2+5n+\log_2 n = O(n^2)$

By definition 2, it states that a function $f(n)$ belongs to $O(g(n))$ if there exists positive constants c and n_0 such that, $f(n) \leq c(g(n))$ for all $n \geq n_0$

We have, $f(n) = 3n^2+5n+\log_2 n$

if, $f(n) = 3n^2+5n+\log_2 n$ and $g(n) = n^2$

So, $3n^2+5n+\log_2 n \leq cn^2 \Rightarrow 3n^2+5n+\log_2 n = 3n^2+5n^2+n^2$

$\Rightarrow 3n^2+5n+\log_2 n \leq 9n^2$ [To establish upper bound]

Now, choosing, $c=9$ and $n_0 = 1$, to write

$$3n^2+5n+\log_2 n \leq 9n^2 \text{ for all } n \geq 1$$

By definition 2, we write that, $3n^2+5n+\log_2 n = O(n^2)$

Using definition 3, states that $f(n) = \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is bounded by positive constants c_1 and c_2, $0 < c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$

$f(n) = 3n^2+5n+\log_2 n$, $g(n) = n^2$

To compute: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2+5n+\log_2 n}{n^2} = \lim_{n\to\infty} \left(3+\frac{5}{n}+\frac{\log_2 n}{n^2}\right)$

$= 3+0+0$ [$\frac{5}{n} \to 0$ as $n\to\infty$, $\frac{\log_2 n}{n} \to 0$ as $n \to \infty$]

Since, 3 is a finite constant, by definition 3

= wrong

Given function, $g(n) = 3n^2+5n+\log_2 n$

By def^n 2, we want to show that $g(n) = O(n^2)$ meaning there exist constants $c > 0$, n_0 such that: $g(n) \leq c f(n)$ for all $n \geq n_0$

where $f(n) = n^2$.

To upper bound each term,

$3n^2$ is already in $O(n^2)$

$5n \leq 5n^2$ for $n \geq 1$, so its in $O(n^2)$

$\log_2 n \leq n^2$ for sufficiently large n

Thus for large n, we can write,

$g(n) = 3n^2+5n+\log_2 n \leq 3n^2+5n^2+n^2 = 9n^2$

for $n \geq 1$, $g(n) \leq 9n^2$

we can choose, $c=9$, $n_0=1$

Hence, by definition 2,

$$g(n) = O(n^2)$$

Definition 3 states, $g(n) = O(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty \rightarrow c$ for some $c \in \mathbb{R}^+$

To compute, $\lim_{n\to\infty} \frac{g(n)}{n^2} = \lim_{n\to\infty} \frac{3n^2+5n+\log_2 n}{n^2} = \lim_{n\to\infty}\left(\frac{3n^2}{n^2} + \frac{5n}{n^2} + \frac{\log_2 n}{n^2}\right)$

$\lim_{n\to\infty} \frac{g(n)}{n^2} = 3+0+0 = 3$ $\left[\frac{3n^2}{n^2} = 3, \frac{5}{n} \to 0 \text{ as } n\to\infty, \frac{\log_2 n}{n^2} \to 0 \text{ as } n \to \infty\right]$

Since 3 is a finite constant, by def^n 3, we conclude, $g(n) = O(n^2)$

2. Tree sort: works by inserting elements into a Binary Search Tree (BST) and then performing an in-order traversal to retrieve the elements in sorted order. Average case time complexity: $O(n\log n)$, In a balanced BST, each iteration takes $O(\log n)$, leading to a total of $O(n\log n)$ for n elements. Worst case time complexity $O(n^2)$ and worst case space complexity is $O(n)$. as storing elements in sorted order the BST degenerates into a linked list, resulting in $O(n)$ space usage.

Odd-Even Transposition Sort: a parallel sorting algorithm that repeatedly compares and swaps adjacent elements in two alternating phases. Average case time complexity: $O(n^2)$ → the algorithm is based on a pairwise comparison and requires $O(n)$ passes in the worst case. Worst case cospace coplexity is $O(1)$ because sorting happens in place without requiring extra memory.

Quicksort: is a divide and conquer sorting algorithm that selects a pivot partitioning the array into elements smaller and greater than the pivot and recursively sorts the subarrays. Average case time copplexity is $O(n\log n)$ performs $O(\log n)$ levels of recursion where each level involves $O(n)$ work. Worst case space complexity $O(n)$ with already sorted input with largest or smallest pivot chosen. can be improved to $O(\log n)$ using optimization.

Jutification: The avg case time complexity of tree sort arises from the expected height of a randomly constructed BST. which is $O(\log n)$. The depth of the tree remains logarithmic, ensuring efficient insertion and traversal.

Odd-even transposition is similar to bubble sort that requires $O(n^2)$ in average case. In the case of quicksort, if the pivot is selected efficiently, it would be $O(n \log n)$

3. (i) Linear insertion sort over binary insertion sort: when sorting a nearly sorted or small array in an online manner. Linear insertion sort has a best case time complexity of $O(n)$ when the input is nearly sorted. Binary insertion sort improves searching time using binary search $O(\log n)$ but the actual insertion still takes $O(n)$. If elements arrive one by one, linear insertion sort is preferable.

(ii) Heapsort over Quicksort: when worst case performance consistency and inplace sorting are required. Heapsort guarantees $O(n \log n)$ worst case time complexity unlike quicksort which can be $O(n^2)$ in worst case. Also, heapsort requires $O(1)$ space where quicksort may require $O(n)$ space.

(iii) Quicksort over Mergesort: When sorting an array in memory with good cache performance. Quicksort is faster in practice due to better cache locality. Mergesort has $O(n \log n)$ complexity always but requires $O(n)$ extra space, making it less suitable for in memory sorting. Quicksort is preferable for general purpose sorting when space efficiency is important.

(iv) Tournament sort over sorting by ranking: when the sorted array needs to be dynamically updated and to find a small subset of top-ranked elements. Tournament maintains a structure where new elements can be inserted and sorted efficiently unlike sorting by ranking which requires complete re-ranking when a new element arrives. mostly used in sports tournaments, gaming leaderboards.

(v) Odd-even over bubble: When parallel processing is possible. Odd-even can be efficiently parallelized. Its suitable for multithreaded architecture. Bubblesort requires $O(n^2)$ even when parallelism is available. as bubble sort is inherently sequencial.

4. Time complexity measures how the running time of an algorithm grows with input size n. It depends on the specific algorithm used to solve a problem. For example, Binary search has $O(\log n)$ time complexity while sequential search has $O(n)$. On the other hand, problem complexity represents the best possible that any algorithm can achieve for a given problem. For example, the problem complexity of searching in a sorted list is $\Omega(\log n)$, meaning no algorithm can do better than $O(\log n)$ in the worst case.

Sequential search checks each element of list one by one. Even in a sorted list, ~~the~~ it takes $O(n)$ in worst case. $A(n) = O(n) \approx n/2$, $W(n) = O(n)$

Jump search involves jumping by a fixed interval and then performing sequential search within a small range. In that case, $W(n) = \frac{n}{i} + (i-1)$ but its $O(n)$ because of fixed i. $\frac{d}{di}(W(n)) = -\frac{n}{i^2} + 1 \Rightarrow 0 = -\frac{n}{i^2} - 1 \Rightarrow i = \sqrt{n}$

Hence, $W(n) = O(\sqrt{n})$ is the best possible with the approach $\left[W(n) = \frac{n}{\sqrt{n}} + (\sqrt{n}-1)\right]$

Recursive partitioning partitions into k intervals of size n/k each. $w(n) = n/i + (i-1) = \frac{n}{n/k} + \frac{n}{k} - 1 = k + \frac{n}{k} - 1 = O(n/k)$. After applying partition recursively we get, $W(n) = k + W(n/k) = k-1 + W(n/k)$. In general, for any fixed k; $W(n) = (k-1)\log_k n + 1$. Binary seach - by dividing the list into two equal halves at each step, binary search significantly improves search efficiency. $\frac{dW}{dk} = \frac{d}{dk}(k-1)\log_k n + 1 = (k-1)\log_e n(-1)(\log_e k)^{-2}\frac{1}{k} + \frac{\log_e n}{\log_e n}$

$= -\frac{k-1}{k}\frac{\log_e n}{(\log_e k)^2} + \frac{\log_e n}{\log_e k} \Rightarrow 0 = \frac{dW}{dk} \Rightarrow \frac{k-1}{k} = \log_e k \Rightarrow \log_e k = 1 - \frac{1}{k} < 1$

$\Rightarrow k < e^1 = 2.7 \Rightarrow k = 2$ [cannot be 1 & greater than 2]

$\frac{d^2W}{dk^2}$ at $k=2$ is ≥ 0 for minimum value. Thus $k=2$ is best. So dividing in more than 2 partition is ~~to~~ not better. $W(n) = k-1 + W(n/k) = 2-1 + W(n/2)$.

$= \log_2 n + 1 \Rightarrow W(n) = \log_2 n + 1$

Decision tree represent the sequence of comparisons needed to search an element in a list. Let, $n = 16$, for sequential, $W(n) = n = 16$; jump $\rightarrow W(n) = 2\sqrt{n} - 1 = 2 \cdot 4 - 1 = 7$, for binary $\rightarrow W(n) = \log_2 n + 1$
$= 4 + 1 = 5$

(1) To ensure a maximum stack depth of $\Theta(\log n)$, while maintaining the $O(n \log n)$ expected running time of the algorithm, we need to make sure to perform the recursive operation on the smaller subarray and iterate on the larger subarray.

The modified TRE-QUICKSORT will be:

```
TRE-QUICKSORT-MOD (A, p, r)
  while p<r:
     q = PARTITION (A, p, r)
     if (q-p < r-q):
        TRE-QUICKSORT-MOD (A, p, q-1)
        p = q+1
     else:
        TRE-QUICKSORT-MOD (A, q+1, r)
        r = q-1
```

This modified algorithm applies recursion on the smaller portion and iterates over the larger one in each iteration. Since, each recursive call operates on half the elements (at most), the depth follows a logarithmic pattern. The runtime will be,

$$O(n \log n) + O(\log n) = O(n \log n)$$

Therefore, keeping the running time at $O(n \log n)$, the worst case stack depth for the modified algorithm will be $\Theta(\log n)$.

5. ~~5~~. Let $D(n)$ be the recursion depth for quick sort on a list of n integers. The recurrence for the maximum stack depth is

$$D(n) = D(n/2) + 1 = D(n/4) + 2 = D(n/8) + 3 \cdots = D(n/2^k) + k$$

The recursion bottoms out when: $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$$D(n) = D(1) + \log_2 k \Rightarrow D(n) = O(\log_2 n)$$

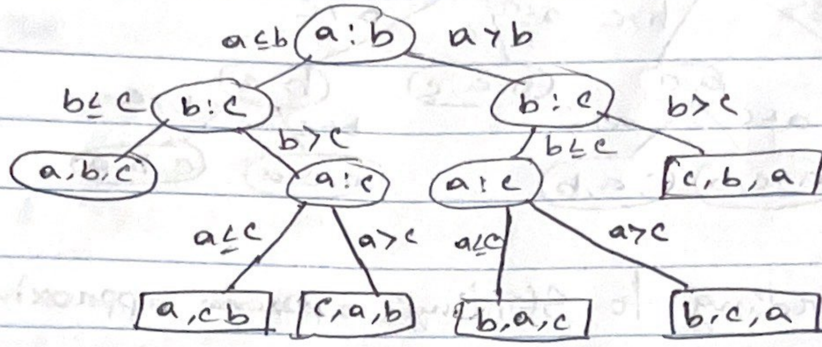
6. Let, an input = a, b, c ; There are 3! outputs (for n numbers there are n! outputs). Every decision tree to sort n numbers must have at least n! leaf nodes. Every decision tree to sort n numbers must have $2n! - 1$ nodes. Every decision tree to sort n numbers must have depth at least $\log_2 n!$ leaf nodes. Now, depth is the lower bound worst case time complexity for this class of algorithms.

$$\log_2 n! = \log_2 (n * (n-1) * (n-2) \cdots 1) = \log_2(n) + \log_2(n-1) + \log_2(n-2) + \cdots + \log_2(1)$$

$$= \sum_{j=1}^{n} \log_2 j$$

$$\leq \int_1^n \log_2 x \, dx = (x \log x - x) \Big|_1^n = n \log n - n$$

Now, Decision tree for 3 numbers.



∴ Depth of binary tree with n! leaves:

$\geq \log_2 n!$

$\geq \int_1^n \log x\,dx$

$\geq (x\log x - x)\Big|_1^n$

$\geq n\log n - n$

Thus, the height of decision tree satisfies $\Omega(n\log n)$. A lower bound on problem complexity of comparison based sorting algorithms is $\Omega(n\log n)$