Exercise 1: Big-$\mathcal{O}$

Consider the following functions:

$$n^{1.5}, n\log n, 2^n, n!, n^n, 1/n$$

Rank the listed functions by order of growth., i.e., give an ordering f_1, f_2, ..., f_5 such that $f_1 = \mathcal{O}(f_2)$, $f_2 = \mathcal{O}(f_3)$, and so on. Justify your ordering.

$$\frac{1}{n} < nlogn < n^{1.5} < 2^n < n! < n^n$$

$$\frac{1}{n} = O(nlogn)$$

$$\lim_{n\to\infty} \frac{1/n}{nlogn} = \lim_{n\to\infty} \frac{1}{n^2 logn} = 0$$

$$nlogn = O(n^{1.5})$$

$$\lim_{n\to\infty} \frac{nlogn}{n^{1.5}} = \lim_{n\to\infty} \frac{logn}{n^{0.5}} = \lim_{n\to\infty} \frac{logn}{n^{1/2}} = \lim_{n\to\infty} \frac{1}{2n^{3/2}} = 0$$

$$n^{1.5} = O(2^n)$$

$$\lim_{n\to\infty} \frac{n^{1.5}}{2^n} = \lim_{n\to\infty} \frac{1.5n}{2^n ln2} = \lim_{n\to\infty} \frac{1.5}{2^n ln^2 n} = 0$$

Proving $2^n = O(n!)$ using Induction:

P(n) : $2^n \leq n!$

Base Case:

P(4) : $2^4 = 16 < 4! = 24$ is true.

Induction Step:

Assume P(n) is true. We want to show P(n + 1) is true.

$$2^{n+1} = 2^n * 2 \leq (n+1)! = (n+1)n!$$

So $2^{n+1} \leq (n+1)!$, for $n \geq 4$, which means P(n + 1) is true.

Thus, 2^n is $O(n!)$. ◻

Proving $n! = O(n^n)$ using Induction:

P(n) : $n! \leq n^n$

Base Case:

P(2) : $2! = 2 < 2^2 = 4$ is true.

Induction Step:

Assume P(n) is true. We want to show P(n + 1) is true.

$$(n + 1)! = (n + 1)n! \leq (n + 1)^{(n+1)}$$

So $n! \leq n^n$, for $n \geq 2$, which means P(n + 1) is true.

Thus, $n!$ is $O(n^n)$. ◻

Exercise 2: Permutations

A permutation of a sequence $(s_0, s_1, \ldots, s_{n-1})$ is a sequence with the same terms, but in a different order. For example, the sequence (3, 4, 6) admits 6 permutations:

(3, 4, 6) (3, 6, 4)
(4, 3, 6) (4, 6, 3)
(6, 3, 4) (6, 4, 3)

1. Give a decrease and conquer algorithm to output all permutations of a sequence of n distinct elements.
2. Prove that this algorithm is correct.
3. Give a recurrence for the runtime of this algorithm.
4. Solve this recurrence.

1. Given the algorithm, we will have a base case where the input size (an array of integers) is 0, then an empty set will be returned. Next, we can store the last element in the array into another variable. Then, we can recursively call the function permutations to divide the array into permutations in a loop from 0 to n - 1, the loop won't include the last element in the array. Finally, we can place the last element in the 0th index to the (n-1)th index in our array to have the final permutation. We will have n! permutations in the end.

2. Proof by Induction

   Base Case:

   n = 0, there is only one permutation, the empty set.

   Induction Hypothesis:

   Assume the algorithm is correct for n elements in the array nums. We will get n! permutations for an array with size n. We want to prove that the algorithm is correct for n + 1 elements with (n + 1)! permutations.

   Inductive Step:

   For an array with size n + 1, the new_element will be the final element, or (n + 1) - 1 = nth element. Since an array with size n contains n! permutations, we can assume the recursive call will be called n * (n - 1) * (n - 2) * … * 1 = n*n! times. Finally, there will be n + 1 positions where the final element can be placed. So, there will be (n + 1)(n)(n!) … (2)(1) permutations. Therefore, we will have (n + 1)n! = (n + 1)! permutations.

Thus, the algorithm is correct for an array with n + 1 elements. ◻

3. The recurrence for runtime will be $T(n) = n * T(n-1)$, where this algorithm makes n recursive calls and each with a sequence of size n - 1.

4. To solve this recurrence:
   $T(n) = n * (n-1) * (n-2) * \ldots * 1 = n!$
   The solution is $T(n) = n!$, representing the time complexity for generating all permutations of a sequence with n distinct elements.

Exercise 3: Hamming Weight

The *Hamming Weight* of a binary number n is the number of bits set to 1 in the binary representation of n. For example, the Hamming Weight of 14 is 3, since

$$14 = 1110_2$$

Give a divide and conquer algorithm to compute the Hamming Weight of a non-negative integer n.

We will have a base case where if n, a non-negative integer, is 0, then the Hamming Weight is 0. We will divide n into 2 halves, the lower bits and the uppermost bit. Then, the algorithm will recursively compute the Hamming Weight of the lower and upper half. Finally, we will combine the Hamming Weights of the lower and upper halves to get the final Hamming Weight.

Exercise 4: Recurrences

Consider the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

1. Show that $T(n) = \mathcal{O}(n^3)$ by induction
2. Solve the recurrence using the DC Recurrence Theorem.

Assume that $T(n) \leq c$ for all $n < n_0$ for some constants c and $n_0 > 0$.

1. Show that $T(n) = O(n^2)$ by induction, not $O(n^3)$

We want to prove that there exists a constant c and an integer K such that $T(n) \leq cn^2$ for all $n \geq k$. We want to show that $T(n) = O(n^2)$ by induction.

Base Case:
$T(1) = 1$

Inductive Hypothesis:
Assume that $T(m) \leq cm^2$ for all m < n. We want to show that $T(n) \leq cn^2$.

Inductive Step:

$$T(n) = 3T(\tfrac{n}{2}) + n^2$$

$$T(k) \leq cK^2 \quad \text{for K < n}$$

$$T(n) = 3T(\tfrac{n}{2}) + n^2$$

$$\leq 3T(c(\tfrac{n}{2})^2) + n^2$$

$$= 3c\frac{n^2}{4} + n^2$$

$$= \frac{3cn^2}{4} + n^2 = n^2(\frac{3c}{4} + 1)$$

$$\frac{3c}{4} + 1 \leq c$$

$$1 \leq \frac{c}{4}$$

$$4 \leq c$$

Therefore, we have shown $T(n) \leq cn^2$ for $c \geq 4$.
Thus, we have proven $T(n) = O(n^2)$. ◻

2. $T(n) = 3T(\frac{n}{2}) + n^2$

DC Recurrence Theorem states that $T(n) = aT(\frac{n}{b}) + f(n)$

$a = 3$

$b = 2$

$f(n) = n^2$

Plug a, b, f(n) into $af(n/b) = cf(n)$.

$3(\frac{n}{2})^2 = cn^2$

$\frac{3}{4}n^2 = cn^2$

$\frac{3}{4} = c \rightarrow c = \frac{3}{4} < 1$

Since c is less than 1, then $T(n) = O(f(n))$.

Therefore, $T(n) = O(n^2)$.

Exercise 5: Sorting

Consider the following sorting algorithm:

```
1: func SELECTION-SORT(arr):
2:     n = LEN(arr)
3:     if n ≤ 1:
4:         return arr
5:     else:
6:         result = SELECTION-SORT(arr[0 : n − 1])    ▷ Sort all but the last element
7:         last-element = arr[n − 1]
8:         let i be the first index such that arr[i − 1] ≤ last-element ≤ arr[i]
9:         result.INSERT(last-element, i)    ▷ Insert last-element into correct position
10:        return result
```

1. Prove the correctness of this algorithm by induction.
2. Write a recurrence, $T(n)$, for the runtime of this algorithm. Assume finding i in line 8 is done via linear search.
3. Solve this recurrence in terms of asymptotic complexity, i.e., find a function $f(n)$ such that $T(n) = \mathcal{O}(f(n))$.
4. How does this recurrence change if we find i via binary search? How does this affect the asymptotic complexity?

1. Show that the algorithm is correct by induction

Base Case:
n = 1, the array only has one element and it's already sorted.

Induction Hypothesis:
Assume the algorithm correctly sorts any array of length n for some n. We want to prove that the algorithm is correct for n+1 elements.

Inductive Step:
The algorithm will sort all of the elements except the last element, on line 6. Then the last element will be inserted into the sorted part of the array at the correct position as on line 9. Thus, with the addition of one more element, the entire array will be sorted.

Therefore, by induction, the algorithm correctly sorts any array including an array with n+1 elements. ◻

2. Recurrence for Runtime (Linear Search)

The runtime of linear search is O(n) as stated on line 8. Then the algorithm recursively sorts n-1 elements as done on line 6. Thus, the recurrence will be $T(n) = T(n-1) + O(n)$.

3. Solve for $T(n) = T(n-1) + O(n)$

Since the algorithm will recursively sort all but the last element, we can solve the recurrence by using these steps:

$$T(n) = T(n-1) + O(n)$$
$$T(n) = T(n-2) + O(n-1) + O(n) = T(n-2) + O(n)$$
$$T(n) = T(n-3) + O(n-2) + O(n-1) + O(n) = T(n-3) + O(n)$$
…
$$T(n) = T(1) + O(2) + O(3) + \ldots + O(n)$$
$$T(n) = O(1 + 2 + 3 + \ldots + n) = O(\tfrac{n(n+1)}{2}) = O(n^2)$$

Thus, the asymptotic complexity of the algorithm with linear search is $O(n^2)$.

4. Impact of Binary Search

The runtime of binary search takes O(nlogn). The recurrence relation of binary search to find i will become $T(n) = T(n/2) + O(logn)$.

The asymptotic complexity will become O(nlogn). The use of binary search will heavily reduce the overall complexity from quadratic to linearithmic.