Seonhye Yang (sy3420)
Fundamental Algorithm (CSCI-GA 1170)

1. $1 \times 1! + 2 \times 2! + \dots + n \times n! = (n+1)! - 1$

Lets test for $n = 1$

$$(n+1)! - 1 = (1+1)! - 1$$
$$= 2! - 1$$
$$= 1$$

Assumption: Assume that the statement holds true for $n = k$

$n = k$

$= 1 \times 1! + 2 \times 2! + \dots + k \times k! = (k+1)! - 1$

- Show that the assumption holds true for $n = k+1$

$1 \times 1! + 2 \times 2! + \dots + k \times k! + (k+1)(k+1)! = (k+1)! - 1 + (k+1)(k+1)!$

$= (k+1)! - 1 + k(k+1)! + (k+1)!$
$= 2(k+1)! + k(k+1)! - 1$
$= (k+2)(k+1)! - 1$
$= (k+2)! - 1$
$= ((k+1)+1)! - 1$

Hence Proved

2. $A = [5, 3, 7, 10, 4]$
$\Rightarrow [3, 4, 5, 7, 10]$ ✓

In general, the best scenario is when array, A, is closest the sorted array or the same as the sorted array. The worst scenario is when array, A, is the reverse of the sorted array.

However with Selection Sort, the time complexity of the best case is the same as the time complexity as the worst case.

Selection Sort is to find the smallest number in an unsorted array and move it to the front of the array. In this problem, Selection Sort finds the largest element and swaps it with the remaining element. But, the process still remains the same.

Time Complexity

The total number of comparison:

$(n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n} i$

$\Rightarrow \sum_{i=1}^{n-1} i = \frac{(n-1)+1}{2}(n-1) = \frac{1}{2}n(n-1) = \frac{(n^2-n)}{2}$, which is complexity $O(n^2)$ in terms of comparison. Each of these scans require one swap for $n-1$ elements.

- Best Case: Time Complexity $O(n^2)$

  $A = [3, 4, 5, 7, 10]$

This is because selection sort needs to look at every element in the array beyound the point where it knows the array has been sorted. Therefore, selection sort takes $\frac{n(n-1)}{2}$ comparsions. But we'll have 0 swaps.

- Worst Case: Time Complexity $\Theta(n^2)$.

  $A = [10, 7, 5, 4, 3]$

This is because selecting the largest element requires scanning all $n=5$ elements in the array. This requires $(n-1)$ comparisons and then swapping. Then finding the next largest element requires scanning all $n-1$ elements and so on: e.g. $(n-1) + (n-2) + \dots + 1$. Each of these scans require one swap for $n-1$ elements.

3. $n!, 3^n, 2^n, n^2, n\log(n), 2^{\log(n)}$

4a. Mathematical Definition:

$f(n) = O(g(n))$ if there exist positive constant $C > 0$, such that, N_0,

$$f(n) \leq c \cdot g(n) \quad \text{for all } N \geq N_0$$

For example: $f(n) = 4n^2 + 10n$

$g(n) = n^3$

Show that $f(n) = O(g(n))$

- We have to find $C > 0$ and a threshold $n_0 > 0$

$C := 14$

$\rightarrow 4n^2 + 10n \leq C \cdot n^3$

$4n^2 + 10n \leq 4n^3 + 10n^3$

$\leq 14n^3 \quad \text{for } n \geq 0$

Now that we understand $f(n) = O(g(n))$, lets prove that if $f = O(g)$, then $g = O(f)$ cannot be true.

The general intuition is that $f(n) = O(g(n))$ when $g(n)$ grows faster than $f(n)$.

$f(n) = O(g(n))$ if $|f(n)| \leq c\,|g(n)|$ for big enough n and constant c.

Counter-example:

$f(n) = n$ and $g(n) = n^2$

- $f(n) = O(g(n)) \Rightarrow n = O(n^2)$ when we take $c = 1$, we would have $n \leq n^2$ for every $n \geq 1$
- However, n^2 can't be $O(n)$ because this implies that $n^2 \leq c \cdot n$ for every $n \geq n_0$ or $n^2 - c \cdot x \leq 0$. This cannot be true because $\lim_{n \to \infty} n^2 - c \cdot n = \infty$

This counter-example suggests that $n = O(n^2)$ but the reverse cannot be true. Therefore, it is FALSE

b. If $f = \Omega(n)$ and $g = \Omega(n)$, then $f + g = \Omega(n)$. TRUE

From the additivity property, this statement is true

$f = \Omega(n)$ where $f(n) \leq c_1 \cdot h(n)$ and $c_1 > 0$ for all $N \geq n_f$

$g = \Omega(n)$ where $g(n) \leq c_2 \cdot h(n)$ and $c_2 > 0$ for all $N \geq n_g$

Then we combine the above:

- $f(n) + g(n) \leq (c_1 + c_2) \cdot h(n)$, $\forall n \geq \max\{n_f, n_g\}$, $c = (c_1 + c_2) > 0$

5. a. 5 reverse pairs:
(2,1), (3,1), (8,6), (8,1), (6,1)

b. MERGE_REVERSE(A, x, y, z):

```
n1 = y - (x+1)
n2 = (z - y)
for i in n1:
    L[i] = A[x+i-1]
for j in n2:
    R[j] = A[y+j]
i = 1
j = 1
reverse_pair = 0
for m = x in z:
    if L[i] ≤ R[j]
        A[m] = L[i]
        i += 1
    else:
        reverse = reverse + n1 - i + 1
        A[m] = R[j]
        j += 1
Return reverse_pair
```

- This sort A[x, y] and returns number of reverse pairs
- L[i] and R[j] track the reverse-pair using i and j.
- This algo returns the number of reverse pairs where i is the index of first sub-array and j is the index of second sub array.
- Merge sort basically divides arrays into sub-arrays and solves them through smaller problems: Recursive.
- This algo is similar to that of merge-sort and the complexity is $\Theta n \cdot \log(n)$