

1. Let's talk about identifying best- and worst-case runtimes. In particular, let's focus on INSERTIONSORT:

```
def InsertionSort(A[0:n]):  
    for i in range(n):  
        j = i  
        while j > 0 and A[j] < A[j - 1]:  
            swap A[j] and A[j - 1]  
            j--
```

First, let's look at the code. The outer for loop *always* has n iterations - **this immediately implies a lowerbound in the runtime of $\Omega(n)$** . The inner for loop always has at least 1 (hence not changing the lowerbound) and at most n iterations. Since all commands take constant time, this gives us an upperbound of $O(n^2)$. In other words, we've established that the best-case runtime is $\Omega(n)$ and the worst-case runtime is $O(n^2)$. Next, let's try some instances:

- Let's start with a sorted list of length n (say $[1, 2, 3, \dots, n]$). On a sorted list, the inner while-loop condition is never true (and hence always takes 0 iterations), **which tells us that the runtime is $\Theta(n)$ on a sorted list**. This tells us that the worst-case runtime is $\Omega(n)$ - technically, we already knew this, because certainly our worst-case runtime wouldn't be faster than our best-case runtime. It doesn't tell us any more than that for worst-case runtime because there could be (and is!) an input on which it takes longer. It *also* tells us that the best-case runtime is $O(n)$ - this input *alone* doesn't tell us any more than that because there could be an input on which INSERTIONSORT takes less time. However, because we know from above that the best-case runtime is $\Omega(n)$ we can immediately determine that the best-case runtime is $\Theta(n)$ - we have obtained a matching upper and lower bound on the best-case runtime.
- Now let's talk about a reverse-sorted list of length n (say $[n, n-1, \dots, 2, 1]$). On a reverse-sorted list, on iteration i of the outer for-loop, the while loop takes i iterations (as it shuffles $A[i]$ down i indices). This tells us that the runtime is $\Theta(\sum_{i=0}^{n-1} i) = \Theta(\frac{(n-1)n}{2}) = \Theta(n^2)$, **which tells us that the runtime is $\Theta(n^2)$ on a reverse-sorted list**. This tells us that the best-case runtime is $O(n^2)$ - this information is superfluous by the above, because we already know that the best-case runtime is $\Theta(n)$, but I'm just pointing it out here for clarity. It *also* tells us that the worst-case runtime is $\Omega(n^2)$ - like above, this input *alone* doesn't tell us any more than that regarding worst-case runtime because there could be an input on which INSERTIONSORT takes more time. However, because we know from above that the worst-case runtime is $O(n^2)$ we can immediately determine that the worst-case runtime is $\Theta(n^2)$ - we have obtained a matching upper and lower bound on the worst-case runtime.

Next, let's show that the algorithm is **correct**: we'll do so by induction *on the iteration i of the outer for-loop*. Specifically, we'll show that at the start of iteration i the first i items in the array are in sorted order.

As a base case, consider when $i=0$: since the first i items are in sorted order (there's no items to even talk about), the statement is vacuously true.

Now, suppose we know that the statement holds for some particular iteration i : we'll show that it still holds for iteration $i+1$. Let's look at what happens on iteration i : we go to $x=A[i]$ and swap x down as long as it is smaller than the element to its left. This means when x stops swapping, either:

- x reached all the way to the left side of the array, in which case nothing is to its left
- the item y to the left of x satisfies $y \leq x$. By induction, everything to the left of y is in sorted order and no bigger than y .

In either case, the array is now in sorted order from the beginning up to x . Additionally, since we swapped our way past them, x must be less than all the items to x 's right, which we haven't changed the order of and thus must still be in sorted order. Thus the array is now in sorted order for the first $i+1$ indices, confirming that our statement holds for iteration $i+1$. This completes the induction.

Because we know that at the start of each iteration i the array has correctly sorted the first i items of the array, we know it's true on the iteration $i=n$, which tells us the algorithm has correctly sorted the first n items.

2. Consider the following problem:

Input: A collection of jobs $J_1, \dots, J_n$ where each job J will take $p[J]$ units of time to complete.

Output: A scheduling of the jobs that minimizes total (or average) completion time.

(a) Determine a brute-force algorithm for this problem.

Solution: Consider the following algorithm:

```
def bruteForce(J[1, ..., n]):
    permutations = GenerateAllPermutations(J)

    bestpermutation = None
    minCost = Infinity

    for permutation in permutations:
        total = 0
        for j in permutation:
            nextCompletion = total + p[j]
            total += nextCompletion
        if total < minCost:
            bestPermutation = permutation
            minCost = total

    return bestPermutation
```

This algorithm tries every possible schedule of the jobs to find the best one. It first runs GENERATEALLPERMUTATIONS, which takes $\Theta(n \times n!)$ time. It then sums up completion times of every job under each possible permutation. As it takes linear time to perform this sum and there are $n!$ different permutations, this for loop takes $\Theta(n \times n!)$ time. Therefore, the total runtime of BRUTEFORCE is $\Theta(n \times n!)$.

(b) Determine a GREEDY algorithm for this problem which fails.

Solution: Consider the algorithm which sorts the jobs by maximal processing time first and the input of two jobs where the processing times are 3 and 5. By scheduling maximal processing time first, the schedule would first run the second job of time 5 and followed by the job of time 3. In doing so, the total completion time would be $5 + (5 + 3) = 5 + 8 = 13$. However, by scheduling the job of time 3 first, the total completion time would be $3 + (3 + 5) = 3 + 8 = 11$, hence showing that the algorithm which prioritizes maximal processing time is incorrect for this problem.

(c) Determine a GREEDY algorithm for this problem which succeeds.

Solution: Consider the algorithm which sorts the job by minimal processing time first. Given some input $J_1, \dots, J_n$, consider **any** valid solution S : we will show that the solution our algorithm produces on the same input is no worse than this one. To do so, scan for the earliest point of disagreement in S from the perspective of our algorithm. If there are no disagreements, then the solution we produce is the same as S , and therefore our solution is worse than S .

If there is a disagreement, suppose it happens when we schedule job j and S schedules a different job j' . Because S and our solution agreed on all previous decisions, S has not scheduled j prior to j' . As a result, j appears later in S . Consider a solution S' that is identical to S except that the positions of j and j' are swapped. By definition of our algorithm, we know that $p[j]$ is the job of minimal processing time remaining. As a result:

- the completion time of j in S' is no greater than the completion time of j in S ,
- the completion times of all jobs between j and j' in S' (which are in the same positions as they were in S) are no greater than they were in S ,
- the completion time of j' in S' is exactly the same as the completion time of j in S ,
- and the completion times of all other jobs in the collection have been unchanged.

As a result, we have produced a solution S' which is no worse than S , but agrees with our solution for an additional step. By induction, we can repeat this process (of resolving disagreements by producing new solutions) until we produce our solution; at every step along the way, our solutions have gotten no worse, showing that our solution is no worse than S .

Since our solutions are always no worse than **any** other solution on the same input, it must be no worse than any **optimal** solution on the same input, showing that our algorithm produces optimal solutions.