

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.

3. Recall the following pseudocode for MERGESORT:

```
def MergeSort(A[0, ..., n - 1]):
    if n <= 1:
        return A

    mid = n / 2
    L = MergeSort(A[:mid])
    R = MergeSort(A[mid:])
    return Merge(L, R)

def Merge(L, R):
    n = len(L) + len(R)
    out = []

    L_idx = R_idx = 0

    for i in range(n):
        if L_idx == len(L) or (R_idx < len(R) and R[R_idx] < L[L_idx]):
            add R[R_idx] to out
            R_idx += 1
        else:
            add L[L_idx] to out
            L_idx += 1
    return out
```

Show that MERGESORT is correct and has a runtime of $\Theta(n \log n)$:

Solution: First, let's establish that the MERGE algorithm works as desired when given two sorted lists L and R. We'll do so by induction on the for-loop; specifically, we'll show that at the start of iteration i:

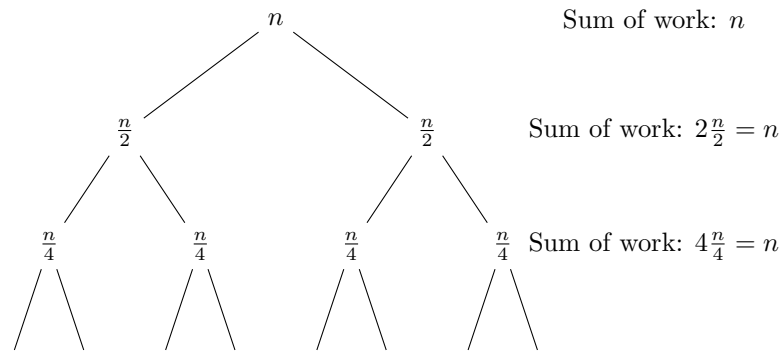
- L[L_idx] is the next smallest item in L that is not yet included in out,
- R[R_idx] is the next smallest item in R that is not yet included in out,
- and out contains the i smallest items from the combination of L and R in sorted order.

These are all vacuously true as a base case, since out is empty at the start of the loop and both L and R are sorted. Now, assume the statement holds for some iteration i - we'll show it holds for i+1. During iteration i, either

- L_idx == len(L) - in which case, there's nothing left in L, so the smallest item left out of L or R is explicitly the next item in R, which we then add to out. By induction, all the previous items in out are smaller than this new item and are already in sorted order, so out is still sorted and now contains the first i + 1 smallest items from L and R. Additionally, since we have included the next smallest item from L we have increased L_idx.
- R_idx < len(R) and R[R_idx] < L[L_idx] - in which case, since both L and R are sorted the next smallest item in R is less than all other remaining items in R **and** all remaining items in L, so we add R[R_idx] to out. We also increase R_idx to the next item in R not yet included in out.
- R[R_idx] >= L[L_idx] - in which case, since both L and R are sorted the next smallest item in L is at most all other remaining items in L **and** all remaining items in R, so we add L[L_idx] to out. We also increase L_idx to the next item in L not yet included in out.

Next, let's establish that the MERGESORT algorithm is correct. Again, we'll do so by induction on n: as a base case, when n <= 1, we immediately return A since an empty list or a list of length 1 are already sorted. Next, let's suppose that for all n up to some value k the algorithm MERGESORT works. Then on a list of length k + 1, we first *recurse on the two halves of the list*: note that each half is of a length $\frac{k+1}{2}$, which is *less than k*! By induction, we know that each half has now been correctly sorted. The final step is to call MERGE on those two sorted halves, which we know from above correctly merges the two halves into one sorted list.

Now for the runtime: for large n , the MERGESORT algorithm first recurses on each half, then spends a linear time merging those two halves. This means the runtime $T(n)$ of MERGESORT on an input of length $n > 1$ is given by $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. To analyze this via a recursion tree, we get:



Because the work per recursive call is decreasing at precisely the same rate at which new function calls are made, the work per level of this tree is always n . Because the input shrinks by a factor of 2 each level, the number of levels L in this tree is given by solving the inequality $n (\frac{1}{2})^L = 1$, which is solved by $L = \log_2(n)$. As such, there are $\Theta(\log(n))$ levels to the tree, each of which we spend $\Theta(n)$ work at, giving us that $T(n) = \Theta(n \log n)$.

4.

Rod Cutting

Input: A length n rod and an array $p[0, \dots, n]$ where $p[i]$ denotes the sell-price of a length i rod.

Output: A collection S of positive integers such that $\sum_{s \in S} s = n$ and $\sum_{s \in S} p[s]$ is maximized.

The difficulty of this problem is the sheer volume of possibilities:

8	7 + 1	6 + 2	6 + 1 + 1
5 + 3	5 + 2 + 1	5 + 1 + 1 + 1	4 + 4
4 + 3 + 1	4 + 2 + 2	4 + 2 + 1 + 1	4 + 1 + 1 + 1 + 1
3 + 3 + 2	3 + 3 + 1 + 1	3 + 2 + 2 + 1	3 + 2 + 1 + 1 + 1
2 + 2 + 2 + 2	2 + 2 + 2 + 1 + 1	2 + 2 + 1 + 1 + 1 + 1	2 + 1 + 1 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1			

The above is all the ways of ‘partitioning’ 8 into a sum of integers. There are 21 such ways - as it turns out, if you let $P(n)$ denote the number of ways to partition n into a sum of integers, you get a function whose growth is exponential. So we can’t try every possible way - we need to be a bit smarter.

A stronger form of what we can calculate is $\text{OPT}(i)$: the best value I can get from a length $i \leq n$ rod. Let’s break it up into cases: if I don’t cut the rod of length i up any further, I would get value $p[i]$ from it. If I make a cut of some length $0 < j < i$, then what I want to know is what the optimal value is on what remains, namely a piece of length j and length $i - j$. Since I want the maximal possible value, I will take the maximum over all these options. Hence we get the following recurrence:

$$\text{OPT}[i] = \begin{cases} 0 & \text{if } i == 0 \\ \max \begin{cases} p[i] & \text{don't make a cut} \\ \text{OPT}[j] + \text{OPT}[i - j] & \text{make a cut at } 0 < j < i \end{cases} & \text{if } i > 0 \end{cases} \quad (1)$$

Note that that inner maximization is over i different options: not making a cut, making a cut into two pieces of sizes 1 and $n - 1$, making a cut into two pieces of size 2 and $n - 2$, and so on.

First, consider the following Python code:

```
def Opt(i):
    if i == 0:
        return 0

    else:
        options = [ p[i] ]
        for j in range(1, i):
            options.append(Opt(j) + Opt(i - j))

        return max(options)
```

Note that for $n > 0$, when I call $\text{Opt}(n)$ I will compute $\text{Opt}(n - 1)$ twice: once when $j = 1$ and again when $j = n - 1$. As a result, runtime $T(n)$ of $\text{Opt}(n)$ is *at least* (actually, much greater than) $2T(n - 1)$, which we have seen before induces an exponential runtime.

So we want to get a better runtime: in particular, for this problem, we can get a quadratic runtime ($\Theta(n^2)$). Consider the following pseudocode:

```
def Opt(n):
    A = copy(p) # so A[i] = p[i]
    for i in range(0, n + 1):
        options = []
        for j in range(1, i):
            add (A[j] + A[i - j]) to options
        A[i] = max(A[i], max(options))
    return A[n]
```

Let's show that after iteration i , $A[j]$ is $\text{OPT}[j]$ as defined above (in equation (1)) for all $j \leq i$. For the base case $i = 0$, at the start $A[i] = p[i] = 0$. Since `range(1, 0)` is empty, this loop doesn't do anything, so `options` remains empty, meaning the final assignment to $A[i]$ doesn't change anything, keeping $A[i] = 0$. As a result, we've handled our base case so that $A[i] = \text{OPT}[i]$.

By induction, assume that after a particular iteration $i - 1$ we have that $A[j] = \text{OPT}[j]$ for all $j \leq i - 1$. On iteration i , originally we have $A[i] = p[i]$; we then calculate `options` which includes all values of $A[j] + A[i - j]$ for all values $0 < j < i$. By induction, all such entries of A are equal to OPT , so `options` includes all values of $\text{OPT}[j] + \text{OPT}[i - j]$ for $0 < j < i$. Since we are taking the maximum over all these options and $p[i]$, we computing the exact same formula as $\text{OPT}[i]$, meaning $A[i] = \text{OPT}[i]$ at the end of the iteration i .