1 Non-overlapping intervals

https://leetcode.com/problems/non-overlapping-intervals/#/description

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.
Note: You may assume the interval's end point is always bigger than its start point. Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.

Example 1: Input: [[1,2], [2,3], [3,4], [1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.

Example 2: Input: [[1,2], [1,2], [1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of intervals non-overlapping.

Example 3: Input: [[1,2], [2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

1.1 Algorithm

One important thing we want you to take away from the named algorithms in class is how to reduce problems to those named algorithms. In this case, our problem can be reduced to the activity selection problem. Choosing a minimum set of activities to be removed is the same as choosing a maximum set of non-overlapping activities.

Thus, it suffices to compute the maximum set of non-overlapping activities, using the methods in the activity selection problem, and then subtract that number from the number of activities.

```
# Definition for an interval.
# class Interval(object):
#     def __init__(self, s=0, e=0):
#         self.start = s
#         self.end = e

import sys

class Solution(object):
```

```
def eraseOverlapIntervals(self, intervals):
    """
    :type intervals: List[Interval]
    :rtype: int
    """
    # Iterate through intervals in order of finish time, choosing an
    # interval if it does not overlap with a previous one
    intervals = sorted(intervals, key=lambda interval: interval.end)
    count = 0
    current_time = -sys.maxint
    for interval in intervals:
        if interval.start >= current_time:
            # Include the interval
            count += 1
            current_time = interval.end

    return len(intervals) - count
```

2 Assign Cookies

Consider the problem at

`https://leetcode.com/problems/assign-cookies/#/description`

> Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child $0 \leq i \leq n-1$ has a greed factor g_i, which is the minimum size of a cookie that the child will be content with; and each cookie $0 \leq j \leq m-1$ has a size s_j. If $s_j \geq g_i$, we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.
>
> Note: You may assume the greed factor is always positive. You cannot assign more than one cookie to one child.
>
> Example 1: Input: [1,2,3], [1,1]
> Output: 1
> Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3. And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content. You need to output 1.
>
> Example 2: Input: [1,2], [1,2,3]
> Output: 2
> Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2. You have 3 cookies and their sizes are big enough to gratify all of the children, You need to output 2.

2.1 Algorithm

We define our subproblems as follows:

Suppose the children and cookies are both sorted in increasing order of value. Let $C_{i,j}$ be the number of children who get matched to a cookie if we only use children $i, i+1, \ldots, n-1$ and cookies $j, j+1, \ldots, m-1$.

If $g_i > s_j$, then cookie j is not a match for child i, and since the children are sorted in increasing order of greed, cookie j is not a match for any larger child either. Therefore, $C_{i,j} = C_{i,j+1}$.

If $g_i \leq s_j$, then the optimal thing to do is to assign cookie j to child i (we will prove this later), so $C_{i,j} = 1 + C_{i+1,j+1}$.

Naively, we may compute $C_{0,0}$ with the following recursive algorithm:

```
def findContentChildren(g, s):
    g = sorted(g)
    s = sorted(s)
    return findContentChildrenHelper(g, s, 0, 0)

def findContentChildrenHelper(g, s, i, j):
    if i >= len(g) or j >= len(s):
        return 0
    elif g[i] <= s[j]:
        return 1 + findContentChildrenHelper(g, s, i + 1, j + 1)
    else:
        return findContentChildrenHelper(g, s, i, j + 1)
```

This algorithm is polynomial time, since each recursive call only produces one recursive subproblem, and there are at most m layers of recursive calls (since j gets incremented by 1 at each level). However, for large inputs, this algorithm will exceed the maximum recursion depth on most computers, so I advise you to use the following equivalent "unrolled" two-pointer solution instead. Note that this solution was accepted on LeetCode, but the recursive solution wasn't.

```
def findContentChildren(g, s):
    g = sorted(g)
    s = sorted(s)
    i = 0
    j = 0
    count = 0
    while i < len(g) and j < len(s):
        if g[i] <= s[j]:
            # Then the cookie is a match
            count += 1
            i += 1
            j += 1
        else:
            # Then the cookie cannot be a match for any future child, so discard it
            j += 1
    return count
```

Both of these algorithms run in $O(k \log k)$, where $k = \max(m, n)$. The largest terms in the runtime bound are the sorting terms.

2.2 Proof of correctness

We will prove the correctness of the recursive version of the algorithm.

Inductive hypothesis: $findContentChildrenHelper(g, s, i, j)$ correctly computes the optimal number of children that are assigned to cookies if we only use children $i, i+1, \ldots, n-1$

and cookies $j, j+1, \ldots, m-1$.

Base case: When $i = n$ or $j = m$, we are either using no children or no cookies, so the correct answer is 0, which is what we return.

Termination: At the top level of the recursion tree, we compute $findContentChildrenHelper(g, s, 0, 0)$. By the inductive hypothesis, this correctly computes the optimal number of children that are assigned to cookies if we use the children $0, \ldots, n-1$ and cookies $0, \ldots m-1$. But that's just the optimal number of children that are assigned to cookies if we use all the children and all the cookies, which is what we were trying to compute. Therefore, the algorithm is correct.

Inductive step: Suppose $findContentChildrenHelper(g, s, x, y)$ is correct for all pairs (x, y) where $x \geq i$, $y \geq j$, and $(x, y) \neq (i, j)$. We need to show it is correct for the pair (i, j).

If $g_i > s_j$, then cookie j is not a match for child i, and since the children are sorted in increasing order of greed, cookie j is not a match for any larger child either. Therefore, no optimal solution may include cookie j, and the optimal solution for (i, j) is just the optimal solution for $(i, j+1)$. Since $findContentChildrenHelper$ correctly computes the optimal solution for $(i, j+1)$ (by the inductive hypothesis), and $findContentChildrenHelper(g, s, i, j) = findContentChildrenHelper(g, s, i, j+1)$, the algorithm is correct in this case.

If $g_i \leq s_j$, we show that there must exist some optimal mapping of children to cookies that maps child i to cookie j. Let OPT be an optimal mapping. OPT must include either child i or cookie j, because otherwise, we could add (i, j) to the mapping and produce a more optimal mapping.

Suppose OPT includes child i, but not cookie j. Then child i is mapped to a different cookie (call it w). Since $s_j \geq g_i$, we can map child i to cookie j instead, and create a mapping that is no less optimal.

Similarly, if OPT includes cookie j, but not child i, we may switch cookie j from a different child to child i, creating a mapping that is no less optimal than OPT.

Finally, suppose OPT includes both child i and cookie j. Then either they are mapped to each other (in which case we are done), or child i is mapped to a larger cookie w, and cookie j is matched to a greedier child v. In that case we can swap stuff around so that child i is mapped to cookie j and child v is mapped to cookie w. This is because we already know that $g_i \leq s_j$, and we know that $g_v \leq s_j \leq s_w$. This creates a solution that is no less optimal.

Therefore, there always exists an optimal mapping of children to cookies that maps child i to cookie j, so to produce such a mapping, we may map i to j, and then optimally map the rest of the children with the rest of the cookies. Since the optimal number of the rest of the children that get mapped with the rest of the cookies is $findContentChildrenHelper(i+1, j+1)$, it follows that the optimal number of children that get mapped if you include i and j is just $1 + findContentChildrenHelper(i+1, j+1)$. Therefore, the algorithm is correct in this case.

3 Another fun reduction

This was going to be on the final exam, but we removed it.

The *integer partition* takes a set of positive integers $S = s_1, \ldots, s_n$ and asks if there is a subset $I \in S$ such that

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i$$

Let $\sum_{i \in S} s_i = M$. Give an $O(nM)$ dynamic programming algorithm to solve the integer partition problem.

3.1 Algorithm

We can directly reduce this to the knapsack problem.

Since the integers are all positive, the integer partition problem should return True if and only if we can find some subset of integers that sum to $M/2$.

Consider an instance of the knapsack problem where each item i has weight s_i and value s_i, and the knapsack has capacity $M/2$. If this knapsack problem returns a value of $M/2$, then we know there is a set of integers that sum to $M/2$, and we can put them on the left side of the partition. If it doesn't return a value of $M/2$, then we know there is no set of integers that sum to $M/2$, and since the integers are all positive, this means the integers are not partitionable.