We have learned that NP-Complete problems are almost certainly not solvable in polynomial time (by which I mean: virtually all researchers believe that we will never find a polynomial time algorithm for any NP-Complete problem). However, NP-Complete problems are all solvable by a simple algorithm: try every possible solution to see if any of them works. Since the number of potential solutions to examine can be exponentially large (for example, a Boolean expression with n literals has $2^n$ possible truth assignments), this Brute Force and Ignorance approach is not a polynomial time algorithm.

Even though we cannot do better than exponential time complexity for solving NP-Complete problems, we can still apply some smarts to improve the algorithms. As an example, we will now examine a way to greatly improve on the BFI algorithm for Subset Sum.

The Subset Sum problem: **Given a set $S$ of n integers and a target value $k$, does $S$ have a subset that sums to $k$?**

The BFI algorithm simply examines every subset of $S$ to see if any of them sums to the target value $k$. Since $S$ has $2^n$ subsets, this algorithm runs in $O(2^n)$ time. (You may wonder why I don't include a time factor for computing the sum of each subset - in fact, the sum of each subset can be computed in constant time. **Exercise: see if you can see how to do this.**
To see how we can improve on this, we first need to consider a much simpler problem.

**Pair-Sum**: Given a set $S$ of $n$ integers and a target integer $k$, does $S$ contain a pair of values that sum to $k$?

Pair-Sum is obviously solvable in polynomial time: we can simply compute the sum of each pair of values in $S$, of which there are $\binom{n}{2} = \frac{n(n - 1)}{2}$, which is in $O(n^2)$

But a better algorithm for Pair-Sum is to start by sorting $S$, then work through the sorted list from both ends, eliminating values when we determine they cannot be in a pair that sums to $k$.

Suppose the sorted set looks like this (drawn as if it is stored in an array)

| $s_1$ | $s_2$ | $\cdots$ | $s_{n-1}$ | $s_n$ |

We start by adding computing $t = s_1 + s_n$. There are three possibilities:
- $t = k$ : in this case we can stop ... we have found a pair that sums to $k$.
- $t < k$ : in this case we know $s_1$ cannot be in a solution – adding $s_1$ together with any other element of $S$ will give a total $< k$.
- $t > k$ : in this case we know $s_n$ cannot be in a solution – adding $s_n$ together with any other element of $S$ will give a total $> k$

Thus after one addition, we either stop with a solution or we eliminate either the smallest or the largest element of the set. We can now continue in exactly the same way on the remaining $n-1$ elements.
In pseudo-code, this algorithm looks like this:

```plaintext
Given S and k:
Sort S  # S is indexed from 1 to n because I don’t like
         # 0-based addressing
         # Sorting takes O(n*log n) time
left = 1
right = n
while left < right:
    t = S[left] + S[right]
    if t == k: Report “Yes” and exit
    elsif t < k: left++
    else: right++
Report “No” and exit
```

The loop executes < n times and each iteration takes constant time, so the algorithm runs in O(n*log n) + O(n) time, which simplifies to O(n*log n).

So we have reduced the O(n^2) time of the naïve algorithm to O(n*log n) for this clever algorithm. It may not seem like much but for large values of n this is a huge improvement.

But we still haven’t seen how to improve the algorithm for the general subset sum problem! Bear with me for one more preliminary problem.

**2-Set Pair-Sum:** Given sets X and Y with n elements in each set, and a target integer k, is there an \( x \in X \) and a \( y \in Y \) such that \( x + y = k \)?

It should be clear that we can solve **2-Set Pair-Sum** in O(n log n) time. We sort both sets, then start by letting \( t = x_1 + y_n \). As before, if \( t = k \) we are done, if \( t < k \) we can eliminate \( x_1 \) and if \( t > k \) we can eliminate \( y_n \).
At last we are ready to attack Subset Sum in all its glory. This very clever method was first described by Horowitz and Sahni.

Given set \( S \) and target integer \( k \):

Split \( S \) arbitrarily into two equal sized subsets \( S_1 \) and \( S_2 \).

# If \( S \) has an odd number of elements, make the split as even as possible.

# It doesn’t matter which of \( S_1 \) or \( S_2 \) is bigger in this case.

# If \( S \) does have a subset \( T \) that sums to \( k \), there are three possibilities:

- all the elements of \( T \) are in \( S_1 \)
- all the elements of \( T \) are in \( S_2 \)
- some elements of \( T \) are in \( S_1 \) and some are in \( S_2 \)

Compute the sums of all subsets of \( S_1 \). Let this set of sums be \( L_1 \)

Compute the sums of all subsets of \( S_2 \). Let this set of sums be \( L_2 \)

if \( k \in L_1 \) or \( k \in L_2 \):
    report “Yes” and stop  # this takes care of the first two possibilities

else:
    # we need to determine if there is a subset of \( S_1 \) that
    # can be combined with a subset of \( S_2 \) to give a sum of \( k \).
    # This is equivalent to asking if there is an \( x \in L_1 \) and
    # and a \( y \in L_2 \) such that \( x + y = k \) ... it is an instance of
    # the 2-Set Pair-Sum problem
    Sort \( L_1 \) into ascending order
    - label the elements \( x_1, x_2, \ldots \)
    Sort \( L_2 \) into ascending order
    - label the elements \( y_1, y_2 \ldots \)
    Let \( left = 1 \) and let \( right = \text{size}(L_2) \)
    while \( left \leq \text{size}(L_1) \) and \( right \geq 1 \):
        \( t = L_1[left] + L_2[right] \)
        if \( t == k \):
            report “Yes” and exit
        elsif \( t < k \):
            # this means that \( L_1[left] \) is too small to be in any
            # solution to the problem
            left++
        else:
            # this means that \( L_2[right] \) is too big to be in any
            # solution
            right--
    report “No”

You should convince yourself that this algorithm correctly solves Subset Sum in all cases, for “Yes” and “No” answers. We now determine its complexity.
Computing the sets $L_1$ and $L_2$ takes $O(2^{(n/2)})$ time since each of $S_1$ and $S_2$ has $\frac{n}{2}$ elements. $L_1$ and $L_2$ each have $2^{(n/2)}$ elements. Sorting each of $L_1$ and $L_2$ takes $O(2^{(n/2)} \cdot \log(2^{(n/2)}))$ time, which simplifies to $O(n \cdot 2^{(n/2)})$. The loop iterates at most $2 \cdot 2^{(n/2)}$ times, doing constant-time work on each iteration.

Thus the dominant step is the sorting of $L_1$ and $L_2$, and the entire algorithm runs in $O(n \cdot 2^{(n/2)})$ time.

This is still exponential but it is way better than the BFI algorithm. This table shows the first few values in the comparison (with $n$ even, to make it easy on my brain).

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \cdot 2^{(n/2)}$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>320</td>
<td>1024</td>
</tr>
<tr>
<td>12</td>
<td>768</td>
<td>4096</td>
</tr>
</tbody>
</table>

What made this work? It was the result of splitting $S$ into $S_1$ and $S_2$, thereby reducing the number of subsets we had to sum from $2^n$ to $2 \cdot 2^{\frac{n}{2}}$ ... and then using the 2-Set Pair-Sum algorithm to eliminate combinations.

Our next unit will focus on problems where dividing the problem into subproblems then combining the solutions leads to efficient algorithms.
Divide and Conquer Algorithms

The Divide and Conquer Paradigm

To solve a problem of size $n$:

If $n$ is "small":

solve the problem directly

else:

Subdivide the problem into two or more (usually disjoint) subproblems
Solve each of the subproblems recursively
Combine the subproblem solutions to get the solution to the original problem

Examples of D&C algorithms are familiar to everyone who has studied computing: binary search, Quicksort, and Mergesort are classic examples.

In this class we looked at a possibly less-familiar application of D&C: determining if a tree contains a path of length $k$ or more. (Note: the length of a path is the number of edges in the path.)

First, let us consider the more general problem: given a graph $G$ and an integer $k$, does $G$ contain a path of length $k$? We can call this the $k$-path problem.

This is the generalized version of a well-known NP-Complete problem: the Hamiltonian Path Problem, which simply asks the question “Does graph $G$ contain a path that includes every vertex of $G$?” Since this problem easily reduces to $k$-path (exercise: work out this extremely simple reduction), we conclude that there is almost certainly no polynomial-time algorithm for $k$-path.

However, the story changes when we restrict the question to trees.
**Definition:** A tree is a connected graph with no cycles. The practical implication of this is that for any pair of vertices in a tree, there is exactly one path of edges that joins them.

In a tree, some vertices are joined by short paths and some by longer paths. If the tree represents a message passing network, the length of the longest path is of interest as it gives some kind of measure of the maximum delay for a message to travel from its source to its destination. If there is a path of length k, some messages will take k steps to get from source to destination.

Determining the existence of a path of length k (or more) is not difficult. In fact we will develop algorithms that solve a more general (and more useful) problem: **given a tree T, what is the length of the longest path in T?**

A simple Breadth-First Search or Depth-First Search algorithm can be used n times, to start at each vertex and compute the lengths of the paths from that vertex to all others in O(n) time (this algorithm has a higher complexity on graphs that are not trees). Since in a tree there is exactly one path between each pair of vertices, we don't have to consider alternative paths between two vertices when looking for the longest path: the length of the longest path is simply the maximum distance between any two vertices of the tree. Finding the distances from each vertex to all the others, and keeping track of the longest distance, gives a simple O(n²) algorithm for finding the longest path in the tree: apply an O(n) search algorithm n times.

But we can do better! Not surprisingly, given the theme of this unit, we will use a Divide and Conquer approach.

Suppose we have a tree T. Choose an arbitrary vertex x as the root of T - we can visualize grabbing vertex x and pulling it upwards, so that the rest of the tree "hangs down" from x. Let $T_1, T_2, \ldots$ represent the subtrees that hang from x.
Now we can make an observation about the longest path in T. It may seem a bit obvious but it is very useful:

Either the longest path in T goes through vertex x ...

... or it doesn't

Trivial as it may seem, this observation motivates our algorithm. Because if the longest path in T doesn't go through x, then it (the path) must be completely contained in one of the $T_i$ subtrees ... and we can find the longest path in each subtree recursively (see how craftily Divide and Conquer creeps in).

However, it is entirely possible that the longest path in T does go through x (remember, we chose x arbitrarily). In this case, the longest path must "come up out of" some $T_i$ pass through x, then go "down into" some other $T_j$. And since this path is the longest possible path in T, we must be using the longest path that goes from the bottom to the top of $T_i$ and the longest path that goes from the bottom to the top of $T_j$ - and in fact, we must be using the longest and second longest of the available “bottom to top” paths in the subtrees.

So for each $T_i$, we need the length of the longest path that is completely contained in $T_i$ and we also need the length of the longest path that goes from the bottom of $T_i$ to the top of $T_i$. From this information, we can compute the length of the longest path in T.

The following algorithm is fully recursive. Given a tree T and the root vertex of T, it computes and returns the two values just described.
Max_Tree_Path(T, x)  // x is the root of T (T may be a subtree of the
original tree)

# We compute two values for T:
#    LP : the length of the longest path in T
#    BT : the longest path in T with one end at the bottom of T
#       and the other end at x (BT stands for “bottom to top”)
# This algorithm will return both of these values · if implemented in
# a language that permits only single value returns, an object
# containing both of these paths must be constructed and returned.

# base case
if T consists of just the vertex x:
    LP = 0      # the longest path in T contains 0 edges
    BT = 0      # the longest path from the bottom to the top of T
        # contains 0 edges
else:
    Let $T_1, T_2, \ldots T_k$ be the subtrees that are attached to x, each
        rooted at the vertex $x_i$ that connects directly to x
    max_subtree_LP = 0
    max_subtree_BT = -1  # Exercise: why do these start at -1
    second_max_subtree_BT = -1
    for i = 1, 2, \ldots k
        L, B = Max_Tree_Path($T_i$, $x_i$)
            # this recursive call returns the LP and BT values for
            # subtree $T_i$
        if L > max_subtree_LP:
            max_subtree_LP = L
        if B > max_subtree_BT:
            second_max_subtree_BT = max_subtree_BT
            max_subtree_BT = B
        elsif B > second_max_subtree_BT:
            second_max_subtree_BT = B
    # now we have all the information we need to compute LP and BT
    BT = max_subtree_BT + 1
    LP = max( BT, max_subtree_LP, max_subtree_BT +
        second_max_subtree_BT + 2 )

return LP, BT

This algorithm has some interesting points. In most D&C algorithms we know
exactly what information we need from each subproblem. In this algorithm we
don't know whether the LP or the BT values from the subtrees will be most
useful to find the overall solution, so we compute both of them for each subtree.

The complexity of the algorithm may look difficult to compute. We can define MTP(T) to be the time required for Max_Tree_Path to run on a tree T, and the recurrence relation would look something like this:

\[ MTP(T) = c + MTP(T_1) + MTP(T_2) + \ldots + MTP(T_k) \]

where \( T_1 \ldots T_k \) are the subtrees. This is hard to analyze because we can't predict the sizes of the subtrees.

However, if we think of the "execution tree" of the recursive calls made by the algorithm, we see that every single vertex of the tree gets to be the root of a subtree exactly once during the execution of the algorithm.

So the number of calls to the function is exactly equal to the number of vertices, and the work related to each recursive call is constant (just a few comparisons and additions). This shows that the complexity is \( O(n) \).

We can also think of this, as I suggested in class, as a process of passing information along the edges of the graph. Each time we finish a recursive call, two values get passed up the edge from the root of the subtree to its parent. So each edge contributes a constant amount to the total work that is done, and since there are \( n-1 \) edges, the total amount of work is \( O(n) \).

Thus the algorithm runs in \( O(n) \) time - a full order of magnitude faster than the \( O(n^2) \) algorithm that we started with. If we are dealing with a very large tree, that's a big advantage.
The existence of such a fast algorithm for solving on trees a problem which is known to be extremely difficult to solve on general graphs leads to an obvious but profound question: **does it only work on trees, or can we use the same technique on other types of graphs?** For example, what if we add one edge to a tree - can we still solve the longest-path problem in polynomial time? (The answer is Yes - **try to figure out how to do it**.) It turns out that there are many families of graphs for which this and other NP-Complete problems can be solved in polynomial time. These results are beyond the scope of this course but they are not difficult to understand. For more information search for "k-terminal recursive graphs".