20190123, 20190125

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:

```
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 S1 and S2.
  #If S has an odd number of elements, make the split as even as possible.
  #It doesn't matter which of S1 or S2 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 S1
#     - all the elements of T are in S2
#     - some elements of T are in S1 and some are in S2

Compute the sums of all subsets of S1.   Let this set of sums be L1
Compute the sums of all subsets of S2.   Let this set of sums be L2

if k ∈ L1 or k ∈ L2:
          report "Yes" and stop      # this takes care of the first two
                                     # possibilities
else:
          # we need to determine if there is a subset of S1 that
          #  can be combined with a subset of S2 to give a sum of k.
          # This is equivalent to asking if there is an x ∈ L1  and
          #      and a y ∈ L2 such that x + y = k … it is an instance of
          #      the 2-Set Pair-Sum problem
          Sort L1 into ascending order
               - label the elements   x1, x2, ...
          Sort L2 into ascending order
               - label the elements y1, y2...

          Let left = 1     and  let right = length(L2)
          while left ≤ length(L1) and right ≥ 1:
               t = L1[left] + L2[right]
               if t == k:
                    report "Yes" and exit
               elsif t < k:
                    # this means that L1[left] is too small to be in any
                    # solution to the problem
                    left++
               else:
                    # this means that L2[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 $O(2^{(n/2)} * log(2^{(n/2)}))$ time, which simplifies to $O(n * 2^{(n/2)})$. The loop iterates at most $2 * 2^{(n/2)}$ 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 $O(n * 2^{(n/2)})$ time.

This is still exponential (some call it sub-exponential because the exponent is < n) 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).

n	$n * 2^{(n/2)}$	2^n
2	4	4
4	16	16
6	48	64
8	128	256
10	320	1024
12	768	4096

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 $2 * 2^{\frac{n}{2}}$ … and then using the 2-Set Pair-Sum algorithm to eliminate combinations.

Some very interesting questions came up in class and after class:

Can we improve the efficiency even more by splitting S into a larger group of smaller sets – such as S_1, S_2, S_3, S_4 each of size $\frac{n}{4}$? This sounds good – the number of subsets we actually look at is reduced to $4 * 2^{\frac{n}{4}}$. But now we have to consider combining subsets from every combinition of S_1, S_2, S_3, S_4 (for example, we need to check all sums containing one value from L_1, one value from L_3 and one from L_4, and all sums containing one value from L_2 and one value from L_4, etc.) This balances out the time we saved by making the sets smaller.

Can we improve the efficiency even more by using the same technique recursively to see if S_1 or S_2 contains a subset that sums to k? Yes we can, but these are not the time-critical steps of the algorithm. The step that looks for a solution involving part of S_1 and part of S_2 will still have the same complexity.

Can we improve the efficiency even more by not only computing the sum of the smallest value in L_1 and the largest value in L_2, but also computing the sum of the largest value in L_1 and the smallest value in L_2? Yes, this lets us eliminate two values on each iteration, which cuts the maximum number of iterations by a factor of 2. However we do twice as much work in each iteration so it balances out.

Does that mean that this algorithm cannot be improved? Not at all!!! This is just the best algorithm I know of for this problem – you could be the person who discovers a better one.

(If you enjoy working on this kind of problem, here is a good one: "Powers of 2" Subset Sum. Given a collection of integers S, in which each element is a power of 2 (repetitions allowed), and an integer k, does S have a subset that sums to k? For example, $S = \{1, 1, 1, 2, 2, 8, 16, 16, 128\}, k = 37$. For this instance the answer is "Yes" because $37 = 1 + 2 + 2 + 16 + 16$. The question is, Is this problem NP-Complete, or can you find a polynomial time algorithm for it?)

Our next unit focuses 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 our next class we will look 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.)

This is a particularly interesting problem because the more general problem "Given a graph G and an integer k, does G contain a path of length k?" is NP-Complete. We will see that by restricting the problem to trees, we can solve it very quickly.