20190924

Subset Sum Problem continued ...

We have seen how to solve the Pair-Sum problem efficiently, 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 A1
Compute the sums of all subsets of S2.   Let this set of sums be A2

if k ∈ A1 or k ∈ A2:
           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 ∈ A1  and
           #      and a y ∈ A2 such that x + y = k … it is an instance of
           #      the 2-Set Pair-Sum problem
           Sort A1 into ascending order
                - label the elements   x1, x2, ...
           Sort A2 into ascending order
                - label the elements y1, y2...

           Let left = 1     and  let right = length(A2)
           while left ≤ length(A1) and right ≥ 1:
                t = A1[left] + A2[right]
                if t == k:
                     report "Yes" and exit
                elsif t < k:
                     # this means that A1[left] is too small to be in any
                     # solution to the problem
                     left++
                else:
                     # this means that A2[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. We now determine its complexity.

Computing the sets A_1 and A_2 takes $O(2^{(n/2)})$ time since each of S_1 and S_2 has $\frac{n}{2}$ elements. A_1 and A_2 each have $2^{(n/2)}$ elements. Sorting each of A_1 and A_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 A_1 and A_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 combination of S_1, S_2, S_3, S_4 (for example, we need to check all sums containing one value from A_1, one value from A_3 and one from A_4, and all sums containing one value from A_2 and one value from A_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 A_1 and the largest value in A_2, but also computing the sum of the largest value in A_1 and the smallest value in A_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: can you find a polynomial time algorithm for this problem?)

The last few minutes of class were spent looking *very briefly* at one more quite clever application of the Divide and Conquer method.

Efficient computation of x^n :

Let x be any number and let n be any positive integer. The naïve method for computing x^n would be something like this:

```
def power(x,n):
      result = x
      for i = 2 to n:
            result = result * x
      return result
```

This uses n-1 multiplications. We can do better by observing the following equalities

$$x^n = x^{\frac{n}{2}} * x^{\frac{n}{2}} \quad \text{when n is even}$$

$$x^n = x^{\frac{n-1}{2}} * x^{\frac{n-1}{2}} * x \quad \text{when n is odd}$$

This is clearly a D&C approach, and it is very efficient because the two subproblems at each level are identical! We only have to solve each of them once.

An example will make everything clear. Suppose we want to compute x^{37}

$$x^{37} = x^{18} * x^{18} * x$$

$$x^{18} = x^9 * x^9$$

$$x^9 = x^4 * x^4 * x$$

$$x^4 = x^2 * x^2$$

$$x^2 = x^1 * x^1$$

$$x^1 = x$$

This uses a total of 7 multiplications instead of the 36 needed by the naïve algorithm. It's pretty easy to see that this algorithm has recurrence relation

$$\begin{aligned} T(n) &= c_1 && \text{when } n = 1 \\ T(n) &= T\left(\frac{n}{2}\right) + c_2 && \text{when } n > 1 \end{aligned}$$

As usual, for simplicity we assume $\frac{n}{2}$ is always an integer – if this is not true, the value of $\frac{n}{2}$ just gets rounded down.

In fact it is the same recurrence relation as we saw for binary search, and it has the same complexity: $O(\log n)$

I offered one last observation about this process, without proof or explanation:

The binary version of 37 is 100101 (32+4+1). If we look at the lines of the solution we see that some of them just square the value from the line below, and others square the value from the line below and multiply by x. We can include the bottom line in the second category because it "introduces an x". Designate the "just square" lines as "0" lines, and the "square and multiply by x" lines as "1" lines. Now we can give a short-hand notation for the computation of x^37 as

x^{37} : 1 line
0 line
1 line
0 line
0 line
1 line

We know the last line is $x^1 = x$ and we can work upwards to reconstruct all the other lines – so we can represent the entire bottom to top process as 100101

But look! 100101 is *exactly* the binary representation of 37

And it turns out that this always works (proving this is a nice exercise). So if we want to compute x^{109}, we just take the binary representation of 109 … which is 1101101 … and our computation looks like

1: $x^1 = x$
1: $x^3 = x^1 * x^1 * x$
0: $x^6 = x^3 * x^3$
1: $x^{13} = x^6 * x^6 * x$
1: $x^{27} = x^{13} * x^{13} * x$
0: $x^{54} = x^{27} * x^{27}$
1: $x^{109} = x^{54} * x^{54} * x$

Test 1 will cover everything up to this point.