20190920

Subset Sum

Later in the course we will look at a class of problems that are generally considered to be extremely difficult to solve. Today we will examine one of those problems.

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?**

S is not necessarily a set in the pure mathematical sense: S is allowed to contain duplicates, whereas in a formally defined mathematical set all the elements must be distinct.

S is an example of what we call a decision problem: The answer for any instance is either "Yes" or "No".

For example, let S = {1,1,3,45,61,10000093} and let k = 47. The answer is Yes because 1+1+45 = 47

Computer scientists believe that Subset Sum is so difficult that it is **impossible** to create an algorithm to solve it that runs in O(n^t) time, for any value of t. Note that such an algorithm would have to solve *all* instances of the problem. It is easy to come up with fast algorithms that solve *some* instances of the problem.

However, we can certainly come up with a slow algorithm that does solve Subset Sum: 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.**)

The reason for bringing up this problem now is to examine whether we can use D&C to improve on the BFI algorithm.

To see how, 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 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, the 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.

The earliest reference I have found for this trick is in a textbook by Horowitz and Sahni. They don't claim it as original but they don't give a source.

This is as far as we got on this problem on Friday – we will finish it on Tuesday.