Lecture 10: Algorithmic Thinking

Algorithmic thinking is a way of getting to a solution through the clear definition of the steps needed – nothing happens by magic. Rather than coming up with a single answer to a problem instance, one develops algorithms that can be used to solve all problem instances. Then, the algorithm can be turned into software code.

We’ll look at some interesting puzzles today and come up with algorithms to solve these puzzles. Then, we’ll write software to solve (generalized versions of) these puzzles!

Nuts and Bolts Puzzle

A handyman has a whole collection of nuts and bolts of different sizes in a bag. Each nut is unique and has a corresponding unique bolt, but the disorganized handyman has dumped them all into one bag and they are all mixed up. How best to “sort” these nuts and attach them to their corresponding bolts?

Given $n$ nuts and $n$ bolts, the handyman can pick a nut and try it with each bolt and find the one that fits the nut. Then, he can put away the nut-bolt pair, and he has a problem of size $n - 1$. This means that he has done $n$ “comparisons” to reduce the problem size by 1. $n - 1$ comparisons will then shrink the problem size to $n - 2$, and so on. The total number of comparisons required is $n + (n - 1) + (n - 2) + ... + 1 = n(n+1)/2$. Can one do better? More concretely, can one split the nuts and bolts up into two sets, each of half the size, so we have two problems of size $n/2$ to work on? This way, if the handyman has a helper, they can work in parallel.
Unfortunately, simply splitting the nuts up into two equal sized piles A and B and the bolts into two equal sized piles C and D does not work. If we group nuts and bolts corresponding to A and C together into a nut-bolt pile, it is quite possible that a nut in A may not fit any bolt in C; the correct bolt is in D. However, what we can do is to pick a bolt, we will call it the **pivot bolt**, and use it to determine which nuts are smaller, which one fits exactly, and which nuts are bigger. We separate the nuts into three piles in this way, with the middle pile being of size 1 and containing the paired nut. Therefore, in this process we have discovered one nut-bolt pairing. Using the paired nut, that we will call the **pivot nut**, we can now split the bolts into two piles, the bolts that are bigger than the pivot nut, and those that are smaller. The bigger bolts are grouped with the nuts that were bigger than the pivot bolt, and the smaller bolts are grouped with the nuts that were smaller than the pivot bolt. We now have a pile of “big” nuts and “big” bolts, all together, and a pile of small nuts and small bolts all together. Depending on the choice of the pivot bolt, there will be a differing number of nuts in the two piles. However, we are guaranteed the same number of nuts and bolts in each pile, and moreover, the nut corresponding to any bolt in the pile is guaranteed to be in the same pile!

In this strategy, we had to make \( n \) comparisons given the pivot bolt to split the nuts into two piles. In the process we discover the pivot nut. We then make \( n - 1 \) comparisons to split the bolts up and add them to the nut piles. That is a total of \( 2n - 1 \) comparisons. Assuming we chose a pivot nut that was middling in size, we have two problems roughly of size \( n/2 \), which we can divide again using roughly \( n \) comparisons to problems of size \( n/4 \). The cool thing is the problem sizes halve at each step, rather than only shrinking by 1. For example, suppose \( n = 100 \). The original strategy requires 4950 comparisons. In the new strategy, using 199 comparisons, we get two subproblems each roughly of size 50. Even if we use the original strategy for each of these subproblems, we will only require 1225 comparisons for each one, for a total of \( 199 + 1225 \times 2 = 2649 \) comparisons. Of course, we can do a recursive Divide and Conquer. (The analysis to show that the comparisons in the new strategy applied recursively grows as \( n \log n \) as compared to \( n^2 \) in the original strategy is beyond the scope of this class.)

Interestingly, this puzzle has a deep relationship with perhaps the most widely used sorting algorithm quicksort. The selection of an arbitrary bolt in the recursive Divide and Conquer strategy can be viewed as a selection of a **pivot element**.

Recall in merge sort, we divide the array up into two equal sized subarrays and this division may be such that there are elements in each subarray that are bigger or smaller than elements in the other subarray. Therefore, after the subarrays are sorted, we need a merge operation to find the correct locations for each element in the original array. Mergesort is not an in-place algorithm, since it needs different array storage for the merge step and subarray sorting. We did look at a version of Mergesort that was in-place, but the merge step was not efficient.
In quicksort, we will spend more time in partitioning the array so the merge step becomes trivial. In particular, suppose we have an array with unique elements:

```
  a  b  c  d  e  f  g  h
```

that we wish to sort in ascending order. We will choose an arbitrary pivot element, say, g. Now we will partition the array into two subarrays where the left subarray has elements less than g, and the right subarray has elements greater than g. The two subarrays are not sorted. We can now represent the array as:

```
Elements less than g  g  Elements greater than g
```

We can sort the left subarray without affecting the position of g, and similarly for the right subarray. Once these two subarrays have been sorted, we merely concatenate the results; there is no sophisticated merge step as in the mergesort algorithm. A simplistic implementation of quicksort is shown in `quicksort.py`, which is not in-place. However, the main advantage of quicksort is that the partitioning step, i.e., going from the original array to the one with g’s location fixed and the two subarrays unsorted but satisfying ordering relationships with g, can be done in place as shown in `quicksort-inplace.py`. This is quite clever code and worth reading carefully and understanding. This code assumes that the pivot is selected as the last element of the array, so in the above array, it would be h.

Quicksort is hard to analyze for algorithmic complexity. This is partly because it is unclear what the sizes of the two subarrays are. If, in our example, g was the largest element in the array, then the left subarray would have \( n - 1 \) elements and the right subarray would have 0! Like in the Nuts and Bolts puzzle, we want to pick an element with medium size. Turns out that quicksort’s worst-case complexity is worse than mergesort, but on average it performs better if a random pivot element is picked or the input is randomized. We will leave such analysis for when you take 6.046!

---

**Anagram Puzzle**

Our last puzzle relates to anagrams. An anagram is a word, phrase, or name formed by rearranging the letters of another, such as *cinema*, formed from *iceman*. Let’s say that we have a large corpus of words and our job is to group all the anagrams together. That is, we need to split the corpus into some number of groups, where each group contains words that are anagrams of each other. One way to think about this is that we sort the words in the corpus so all the words that are anagrams of each other are placed next to each other. Given:
We want:

```
  ate  but  eat  tub  tea
```

We could use the following strategy:

```
for each word v in list
  for each word w ≠ v in list
    Check if v and w are anagrams
    If so, move w next to v
```

However, this is quite inefficient, being a doubly nested loop. Is there a better way? There is, using the concept of hashing which is central to the dictionary data structure of Python.

We can compute the **hash** of a string by assigning a unique number to each character and computing some function over these numbers. Typically, this function is multiplication.

```
hash('ate') = h('a') * h('t') * h('e') = 2 * 71 * 11 = 1562
hash('eat') = 1562
hash('tea') = 1562
```

For this hash function, all anagrams will definitely have the same hash! So if we sort the words in the corpus based on what the hash values are, all the anagrams should be grouped together in the sorted corpus. We still have one problem though: two words that are not anagrams may end up with the same hash. For example, if h('m') happens to be 781, the word 'am' will also have a hash of 1562. The word 'am' may appear in between 'ate' and 'eat' in our sorted corpus 😞

Luckily, this problem is easily solved. We will use prime numbers (as we did above) for the hash values corresponding to each letter. Given that each number has a unique prime factorization, the above problem will not occur. Note that this disallows h('m') being 781 because 781 = 11 * 71 is not a prime.

To summarize, the efficient strategy to solve the anagram puzzle is simply to compute a hash of each word by assigning a unique prime to each letter of the alphabet and taking the product. We then sort the words based on the hashes resulting in all anagrams being grouped together in the sorted output. See the code in `anagrams.py`.