Strategy

We have already seen a few strategies for creating algorithms to solve problems.

- The simplest one is the "Brute Force" approach which runs through all the (finitely many) possibilities until we find our solution.
- The most sophisticated one we have seen so far is the "Divide and Conquer" approach where we break each problem into non-overlapping sub-problems of roughly equal size. Iteratively, these can be solved and combined to give the final solution.

There are many other approaches which we shall look at now.

Greedy Method

The basic idea of this method is to look at "locally optimal" approaches. While this may not lead to the optimal solution, it can often lead to *a* solution within reasonable use of resources.

A typical example is used by most people while paying a certain sum of money. Assume that we have a large store of notes of various denominations. How do we pay a certain sum of money using the *least* number of notes? Let $\underline{d} = (d_1, \ldots, d_n)$ be the denominations of notes available and assume they are in decreasing order. Given an amount a to be paid to be paid is divisible by the smallest note available (d_n). We want to find *non-negative integers* m_i such that $\sum_i m_i d_i = a$. Moreover, we want $\sum_i m_i$ to be as small as possible.

The idea is to use as many of the largest notes that will "fit" inside the desired amount. Then for the remaining amount, we repeat this process with the next largest note, and so on.

define `pay`$(a, \underline{d})$
    Set n to be the length of d.
    Check that a is divisible by d_n.
    for *i going from 1 to n.*
        Set c_i to be $\lfloor a/d_i \rfloor$.
        Decrement a by $c_i \cdot d_i$.
    return $(c_1, \ldots, c_n)$

This simple algorithm does produce the optimal answer for this problem in many situations, but not always!

For example, suppose we try to write 8 in terms of $(5, 4, 2, 1)$, then this algorithm will give the answer $(1, 0, 1, 1)$ which uses 3 coins, but the answer $(0, 2, 0, 0)$ which uses 2 coins is better!

There is a lot of theoretical work on studying/identifying problems where the greedy approach gives an optimal answer, or an answer that differs from the optimal one by a bounded error.

Dynamic Programming

The idea is to break a problem into sub-problems which can be compared with each other (partially-ordered) so that the *number* of sub-problems that needs to be solved does not grow exponentially. Note that, unlike the divide-and-conquer approach, the sub-problems are not (necessarily) a fraction of the size of the original problem. In other words, there should be significant overlaps in the sub-problems when considered recursively.

For example, consider the problem of calculating the n-th element in a recursion like the Fibonacci recursion: $F_0 = 0; F_1 = 1; F_{n+2} = F_{n+1} + F_n$ for $n \geq 0$. It is not difficult to see that the number of steps (additions) required to calculate F_n in a brute force way is F_{n-1}. Since F_n grows exponentially, this means that this approach takes $O(c^n)$ to calculate the n-th Fibonacci number.

The calculation of F_{n+2} depends on the knowledge of F_{n+1} and F_n, However, the calculation of F_{n+3} only depends on the knowledge of F_{n+2}, F_{n+1} and F_n. More generally, the calculation of F_{n+k} only depends on the knowledge of F_{h+m} for $0 \geq m < k$. Thus, we *can* reduce the number of steps used if we *store* the sub-problems already solved for re-use! The following Python code explains how this can be done.

```
known = [0, 1] + [-1]*N
def fibo(n):
    if known[n] == -1:
        known[n] = fibo(n-1)+fibo(n-2)
    return known[n]
```

Here N is the largest number for which we may wish to compute F_N. A more elegant approach that avoids such arbitrary upper bounds uses Python "mapping" data types and Python's "exception" handling mechanism is given below.

```
known = {0:0, 1:1}
def fibo(n):
    try:
        return known[n]
    except:
        knwon[n] = fibo(n-1) + fibo(n-2)
        return known[n]
```

In both cases, the answer is obtained in n steps as each value F_k for $k \leq n$ is calculated *at most once*.

Note that this reduction in time complexity requires us to use storage instead. However, that storage is also linear. This is often the case in dynamic programming problems.

Faster Powers. However, in the case of Fibonacci numbers (or similar recursions), there is an even faster method to calculate the values. We prove

that

$$\begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

In fact, for $n = 0$ since $F_2 = F_1 + F_0 = 1 + 0 = 1$. We note that

$$\begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n+2} + F_{n+1} & F_{n+2} \\ F_{n+1} + F_n & F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+3} & F_{n+2} \\ F_{n+2} & F_{n+1} \end{pmatrix}$$

which then proves the result by induction on n.

For any "powering" operation, there is a way to calculate the n-th power in $\log_2(n)$ steps. Let `mul` denote the underlying multiplication operation and **1** denote the identity element for it.

define pow(a,n):
  if n is 0
    return 1
  Set b *as* `mul`(a, a).
  Set c *as* `pow`$(b, \lfloor n/2 \rfloor)$.
  if n mod 2 *is* 0.
    return c
  else
    return `mul`(a, c)

We can use this to calculate the n-th power of the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. This gives the calculation of calculation of F_n in $O(\log(n))$ steps.

An important point to note is that this approach to calculating F_n does so *wihout* calculating F_m for *all* m from 0 to n. This is a feature of algorithms that calculate the value of a function $f(n)$ in $O(\log(n))$ steps. For large enough n, such an algorithm *cannot* be calculating all $f(m)$ for $0 \leq m \leq n$ as doing that would take at least n steps (at least 1 step for each $f(m)$)!

Prune and Search

In the `select` algorithm, we were able to reduce the problem size by a constant factor in each iteration. The "Prune and Search" strategy "prune" the problem size from n to pn for some fixed $p < 1$ in each iteration until we reach some "base cases" where we can easily calculate the solution.

Branch and Bound

A related, but slightly different approach is to create a number of sub-problems ("branch"-ing) and then eliminating those sub-problems where the size parameter ("bound") makes it impossible for the solution to be found in that sub-space.

Backtracking

In some cases, the *only* approach is to keep trying various paths until the right one is found! This is a bit like "brute force" in contexts where there is some structure to the solution space.

A typical example is the 8 Queens problem. The problem is to find 8 positions on an 8×8 chess-board so that no 2 positions are along the same diagonal, anti-diagonal, horizontal or vertical. Obviously, we can ask the same question with 8 replaced by a different size.

The backtracking approach would be as given in the algorithm below. Note that at any stage in Step 1, the following statements hold:

- We are currently searching for a position for the i-th queen. For $1 \leq j < i$ we have found positions p_j which do not share a diagonal, anti-diagonal, horizontal or vertical with each other.
- For $j \leq i$, the set P_j consists all the possible positions for the j-th queen that have not yet been ruled out.

In particular, if P_i is empty (for $i \geq 2$) then we need to backtrack and rule out the previous choice of p_{i-1} from the set P_{i-1}. If P_1 is empty, then no solutions are possible.

0. Start with $i = 1$ and P_1 to be the set of all positions on the board.
1. If $i = 9$ then print the positions and stop.
2. ($i \leq 8$) If P_i is not empty go to Step 3, otherwise go to Step 4.
3. (P_i is non-empty) Carry out the following sequence of steps:
    a. Choose any position in P_i as the i-th position p_i.
    b. Set P_{i+1} to be P_i minus all positions that share a diagonal, anti-diagonal, horizontal or vertical with p_i.
    c. Increment i and go to Step 1.
4. If $i > 1$ and then go to Step 5, otherwise go to Step 6.
5. (P_i is empty and $i > 1$) Carry out the following steps:
    a. Decrement i
    b. Remove p_i from P_i and go to Step 1 (backtrack).
6. (P_1 is empty) Declare that there are no solutions and stop.

Note that we can replace 9 by $n + 1$ in Step 1 to work with an $n \times n$ chess board. In that case, we will also need to ensure that in Step 3b we calculate the diagonal, anti-diagonal, horizontal and vertical for such a board.

We can follow a similar approach for the "Knight's Tour" and the "Knight's Cycle" problems. A similar method can also be used to solve a Sudoku puzzles.