Dynamic Programming

Dynamic programming is an optimization recursion. More often recursion has repeated tasks that are easily cached. Dynamic Programming takes advantage of this by caching anything we can.

There are two types of Dynamic Programming: ***Tabulation*** and ***Memoization***.

How do we build a Dynamic Programming Algorithm?

1. Create a recursive solution
    - Specify what you want to solve. Not how, but what.
    - Create a concise formula on how to solve simple versions of the problem
2. Create the bottom up approach.
    - Start with a base case and work your way up.
    - How can the algorithm call it self? What are the subproblems?
    - Choose a memoization structure that can store the subproblems. *Usually a table (2d array)*
    - Verify evalulation order, make sure all sub problems go after one another.

Tabulation

Tabulation is the bottom-up approach to dynamic programming. Where we start at the base case and move to the end. We fill the array in order as we go.

123^3

Example:

```
def tabFib(n: number) -> number:
        # Base cases
        F[0] = 0
        F[1] = 1

        # Build array
        for i to len(n):
                F[i] = F[i - 1] + F[i - 2]

        return F[n]
```

Memoization

Memoization is similar to tabulation, but using recursion you only build the ones you need and cache called top-down. By building a cache we can reduce the time and not repeat ourselves.

Example:

```
def memFib(n: number) -> number:
        # Base cases
        if n == 0 || n == 1:
                return n

        # Build required items only + cache
        if not F[n]:
                F[n] = memFib(n - 1) + memFib(n - 2)
                return F[n]
```

This example can be done with a dictionary as well.

Text Segmentation

The problem is $splitable(i) \iff A[i..n]$ we want to know if a string is splitable into words. This problem is solvable with backtracking however there are many repeated problems. This is a perfect problem to solve with Dynamic Programming.

Dynamic Solution:

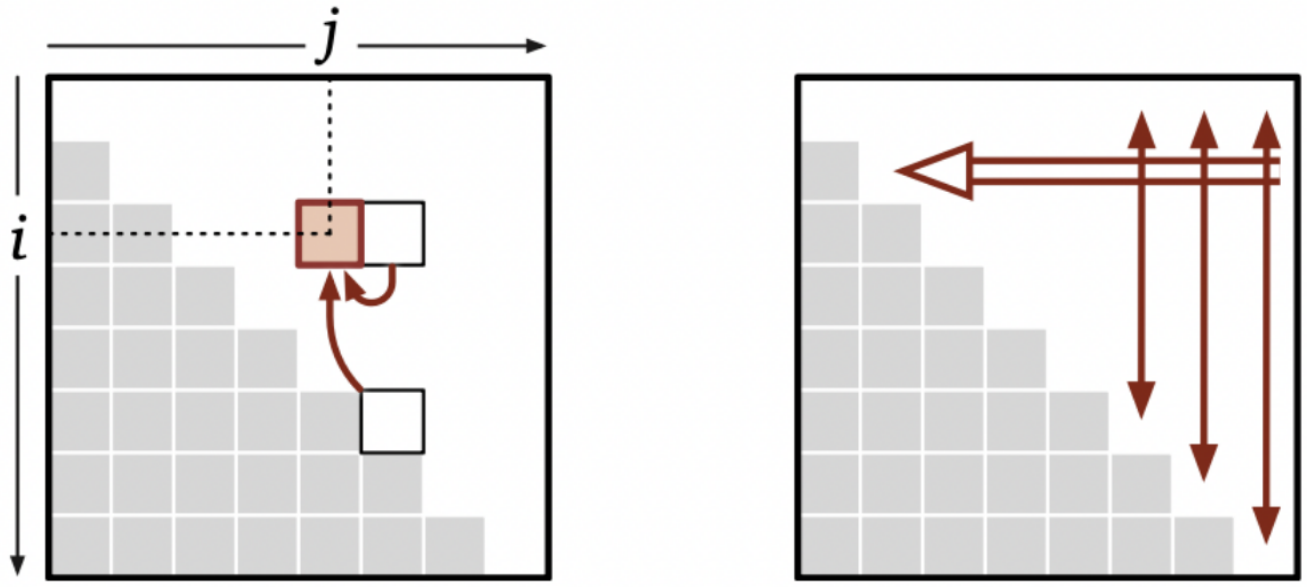
```
# String Array 1 -> n (size)
def Splitable(A[1..n]: list) -> bool:
        # define starting item
        splitTable[n + 1] = True;

        for i in reversed(range(1, n)):
                splitTable[i] = False #default false

                for j in range(i, n):
                        # currently a word and previous word is a word
                        if isWord(i, j) and splitTable[j + 1]:
                                splitTable[i] = True
        # All the words from right and left has to be true for [1] to be true
        return splitTable[1]
```

Longest Increasing Subsequence

Leetcode
Neetcode

The problem is want to find the longest increasing subsequence $A[1..n]$. This is problem we can solve with backtracking. The nature of this problem requires you to recalculate the previous subsequences. A perfect

problem to solve with Dynamic Programming.



The image above shows that we have a reoccuring subproblem that we can save. Each $L(i, j)$ requires $L(i, j+1)$ and $L(j, j+1)$.
The subproblems go right to left. The left is dependant on the right. With this idea we solve right first and go backwards. $R \rightarrow L$

Dynamic Solution:

```
# l[i][j] is a 2d array based on the image above

def Lis(A[1..n]: list) -> int:
        # Create a low base to start increasing subsequence
        A[0] = -999999

        # Zero out memo
        for i in range(0, n):
                l[i, n + 1] = 0

        # Right to left
        for j in reversed(range(1, n)):
                # i navigates from the end
                for i in range(0, j - 1):
                        keep = 1 + L[j, j + 1]
                        skip = L[i, j + 1]

                        # You verify the number is currently smaller than the later
                        if A[i] >= A[j]:
                                L[i, j] = skip
                        else:
                                L[i, j] = max(keep, skip)

        return L[0, 1]
```

Personal Dynamic Solution *(Leetcode)*

```
def Lis (nums: list[int]) -> int:
        n = len(nums)

        # Initialize memo assume 1 LIS
        L = [1] * n

        # Loop through list
        for curr in range(n):
                # Loop previous numbers
                for prev in range(i):
                        # If previous is valid and LIS is less
                        if nums[prev] < nums[curr] and L[curr] < L[prev] + 1:
                                L[curr] = L[prev] + 1

        # Return max LIS
        return max(L)
```

Edit Distance

Leetcode
Neetcode

Given `word1` and `word2` we want to find the minimum about of moves to convert word 1 to word 2.

Our possible moves are

- Insert a character
- Delete a character
- Replace a character

Personal Dynamic Solution:

```
def minDistance(word1: str, word2: str) -> int:
        m = len(word1)
        n = len(word2)

        # Base cases
        if m == 0:
                return n
        if n == 0:
                return m

        # Cache array size m + 1 and n + 1
        dp = [[-1] * (n+1) for _ in range(m+1)]
```

```
        # Fill out minimum operations
        for j in range(n+1):
                dp[m][j] = n - j
        for i in range(m+1):
                dp[i][n] = m - i

        # Build cache bottom up
        for row in range(m - 1, -1, -1):
                for col in range(n - 1, -1, -1):
                        if word1[row] == word2[col]:
                                dp[row][col] = dp[row+1][col+1]
                        else:
                                dp[row][col] = min(dp[row+1][col], dp[row][col+1],
dp[row+1][col+1]) + 1

        # dp[0][0] is the minimum number of items
        return dp[0][0]
```

DP Table Example:

	A	C	D	""
A	1	2	2	3
B	2	1	1	2
D	2	1	0	1
""	3	2	1	0

---

53. Maximum Subarray

Given an int array we want to find the subarray with the largest sum and return its sum.

My personal solution is the window pane method. We move in chunks, if it's ever becomes negative we move on.

Personal Solution:

```
def maxSubArray(nums: List[int]) -> int:
        # Initial max sum is starter
        maxSum = nums[0]
        cur = 0

        for i in range(len(nums)):
                # if negative omit
                if cur < 0:
                        cur = 0

                # Add the current item
```

```
                cur += nums[i]
                # Take the largest number currently
                maxSum = max(cur, maxSum)

        return maxSum
```

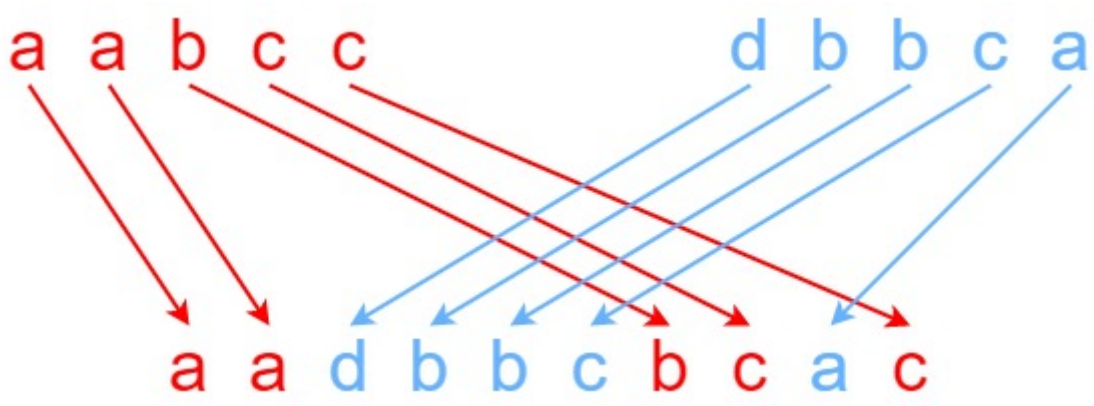
---

97. Interleaving String

Given `S1`, `S2`, and `S3` our goal is to interleave S1 and S2 to create S3. Splitting S1 and S2 into substrings and merge them to create S3. The order of the substrings has to follow the original string.

$s = s_1 + s_2 + \ldots + s_n$
$t = t_1 + t_2 + \ldots + t_m$

Interleave to create:
$s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + \ldots$
or
$t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + \ldots$



Personal Solution:

```
def isInterleave(s1: str, s2: str, s3: str) -> bool:
        # Handle case
        if len(s1) + len(s2) != len(s3):
                return False

        # Create cache
        cache = [[False] * (len(s2) + 1) for _ in range(len(s1)+1)]
        # Define base case
        cache[len(s1)][len(s2)] = True


        for i in range(len(s1), -1, -1):
                for j in range(len(s2), -1, -1):
                        # if the current s1 is equal and there's a path
                        if i < len(s1) and s1[i] == s3[i+j] and cache[i + 1][j]:
```

```
                                cache[i][j] = True
                        # if the current s2 is equal and there's a path
                        if j < len(s2) and s2[j] == s3[i+j] and cache[i][j+1]:
                                cache[i][j] = True

        return cache[0][0]
```

DP Table Example
s1: aab
s2: dbb
s3: aadbbb

	a	a	b	""
d	T	T	T	F
b	F	F	T	T
b	T	T	T	T
""	F	F	T	T

---

647. Palindromic String

Given a string `s` find the number of palindromic substrings.

Personal Solution:

```
def countSubstrings(s: str) -> int:
        count = 0

        for i in range(len(s)):
                # Odd Palindromes
                l = i
                r = i
                while l >= 0 and r < len(s) and s[l] == s[r]:
                        count += 1
                        l -= 1
                        r += 1

                # Even Palindromes
                l = i
                r = i + 1
                while l >= 0 and r < len(s) and s[l] == s[r]:
                        count += 1
                        l -= 1
                        r += 1
```

```
        return count
```