1. **Coin Change** Let's say you have at your disposable a wide assortment of (not neccessarily dollar) coins and you need to make change for a particular value x. As you're doing so, you wonder to yourself *how many different ways can I make chnage for x*. Well I think that's a excellent question so let's figure it out.

   Problem: You are give an integer value x and an array A where each element of the array represents a coin denomination:

   Example: $A = [1, 2, 3]$ and $x = 5$. Output is 5 ($\{1, 1, 1, 1, 1\}, \{1, 1, 1, 2\}, \{1, 1, 3\}, \{1, 2, 2\}, \{2, 3\}$).

Solution: We can solve this problem using a Dynamic Problem approach by initializing an array called $ways$ that is indexed from 0 to x. Each index of the array represents the target amount we want to make from 0 to x and the corresponding array value will represent the number of ways we can make the target amount with the given denomination of coins (A).

Initially, we will set $ways[0]$ to 1 because we can sum up to 0 with the given set of coins in exactly 1 way, which is by using zero coins and the remaining values will be set to 0. For every coin c in A, we will go through $ways$ from index c to x and add to the value at that index the value of $ways[index - c]$. We start from c because we can not use that coin for summing up to any of the index before itself.

$$ways[index] = \begin{cases} 1 & \text{if } index = 0 \\ ways[index] + ways[index - coinValue] & \text{if } index \geq coinValue \\ ways[index] & \text{otherwise} \end{cases}$$

```
CoinChange(coins[1..n], sum):
  for j ← 1 to sum
      ways[j] ← 0
  ways[0] ← 1
  for i ← 1 to n
      for j ← coins[i] to sum
          ways[j] ← ways[j] + ways[j − coins[i]]
      return ways[sum]
```

The runtime complexity of the algorithm will be $O(n \times sum)$ ∎

2. **Subset sum problem**: You are given an array A of size **n** and a number **m** and we have to find whether there exists a subset with sum divisble by **m**.

   Example: $A = [7, 4, 6, 3]$.

   - There exists no subset divisible by 12
   - There exists a subset that is divisible by 8 ($\{7, 3, 6\}$)

Solution: Let $MSum(i, s)$ denote modular sum. If $MSum(i, s)$ is True, then there exists a subset with sum 's' divisible by m at index i. This function obeys the following recurrence:

$$MSum(i, s) = \begin{cases} True & \text{if } i = n \text{ and } s! = 0 \text{ and } s \mod m = 0 \\ False & \text{if } i = n \text{ and } (s = 0 \text{ or } s \mod m! = 0) \\ MSum(i+1, s + A[i]) \,||\, MSum(i+1, s) & otherwise \end{cases} \tag{1}$$

Let $MSum$ be the dynamic programming table, which is a boolean array with m elements. If $MSum[i]$ is True, then there exists a subset whose sum leaves the remainder i when divided by m. We will keep on taking the mod of sum and if at any point $MSum[0] = True$, we can be certain that a subset exists whose sum is divisible by m. The approach is as follows: If we have some subsets with sum = j, we can create a new subset with sum = $(j + A[i]) \mod m$ where $A[i]$ is the current element.

Also when n > m there will always be a subset with sum divisible by m (By pigeonhole principle). So we need to handle only cases where $n <= m$ .The pseudcode for the algorithm is given below:

```
MSum(A[1..n], m):
  if(n > m)
      return True
  for i ← 0 to m − 1
      MSum[i] ← False
  for i ← 1 to n
      if MSum[0] = True
          return True              ⟨⟨Return as soon as we see a sum divisible by m⟩⟩
      Temp[m]                      ⟨⟨Declare boolean Temp array⟩⟩
      for j ← 0 to m − 1
          Temp[i] ← False
      for j ← 0 to m − 1
          if MSum[j] = True
              if MSum[(j + A[i]) mod m] = False
                  Temp[(j + A[i]) mod m] ← True
      for j ← 0 to m − 1
          if Temp[j] = True
              MSum[j] ← True
      MSum[A[i] mod m] ← True     ⟨⟨A[i] mod m is one of the possible sums⟩⟩
  return MSum[0]
```

We have to solve the problem only when n<=m. Thus the upper bound for n is m. So, the resulting algorithm runs in $\boldsymbol{O(m^2)}$ time. ■

3. **KnapSack Problem**: This problem describes a situation where you have a bunch of items that have a corresponding weight and value and your goal is to fit a collection of items with the greatest value into a "knapsack" with a finite capacity.

   So let's formalize this problem: you are given:

   - a array of values V where each element corresponds to item i with value $V[i]$
   - an array of integer weights W where each elements corresponds to item i with weight $W[i]$
   - a integer X which corresponds to the capacity of the knapsack.

   Problem: Find maximum value of items that can be fit into knapsack of the defined capacity.

Solution: Let there be n items, (i.e. $V[1,...,n]$,$W[1,...,n]$). We define the function $Sack(i,Y)$ which is the maximum value of items that can fit in a sack with capacity Y, with items $i,...,n$. If the capacity can contain item i we can then choose if we include item i or not. If we include item i then we increase the value and decrease the capacity accordingly then move on to item $i+1$. If we do not include item i then we move on to item $i+1$. This yields the following recurrence relation:

$$Sack(i,Y) = \begin{cases} 0 & i > n \\ Sack(i+1,Y) & W[i] > Y \\ \max\{V[i] + Sack(i+1, Y - W[i]), Sack(i+1,Y)\} & W[i] \leq Y \end{cases} \tag{2}$$

To implement this we use memiozation.

```
Sack(n,X):
    for k ← 1 to X
        Sack[n+1,k] ← 0
    for k ← 1 to n
        Sack[k,0] ← 0
    for i ← n down to 1
        for Y ← 1 to X
            if W[i] > Y
                Sack[i,Y] ← Sack[i+1,Y]
            else
                Sack[i,Y] ← max{V[i] + Sack[i+1,Y−W[i]], Sack[i+1,Y]}
    return Sack[1,X]
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

■

4. **Largest Square of 1's** You are given a $n \times n$ bitonic array A and the goal is to find the set of elements within that array that form a square filled with only **1**'s.

$$
\begin{array}{cc}
 & i \longrightarrow \\
j \downarrow & \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\end{array}
$$

Figure 1. Example: The output is the sidelength of the largest square of 1's (4 in the case of the graph above, yes there can be multiple squares of the greatest size).

Solution: HW Problem ■

5. **Maximum rectangle**: You are given a 2D array A that contains positive and negative integer values. You need to find the rectangle that has the largest sum of elements.

$$\begin{array}{cc} & i \rightarrow \\ j \downarrow & \begin{bmatrix} 2 & 1 & -10 & 3 & -4 \\ 10 & -6 & 6 & 5 & 4 \\ -1 & 0 & 9 & -5 & -9 \\ -8 & -2 & 7 & 8 & -3 \\ -7 & -2 & 6 & 0 & 4 \end{bmatrix} \end{array}$$

Figure 2. Example: The output is the sum of the greatest rectangle sum (*30* in the case of the array above.).

Solution: Intutively, we know that to find the rectangle with largest sum of elements, we need to compute the sum for all possible rectangles in the 2-D array and compare them all to find the largest sum.

$$\text{Maxsum}_{left,right} =$$

$$\max_{1 \le i \le rows} \left\{ sum[i] = \begin{cases} 0 & sum[i-1] + rowSum[i] < 0 \\ sum[i-1] + rowSum[i] & \text{otherwise} \end{cases} \right\} \tag{3}$$

Step 1: The left and right border of our rectangle can be computed by iteratively fixing a left column from 1 till the number of columns in the array and for every such fixed left column we can set the right column to range from the fixed column till the end to ensure that we cover all possible column ranges.

Step 2: Now for each of these left-right column bounds, in order to find the top and bottom bounds for our rectangle, we compute the sum of the values for each row ensuring that we only take the values that lie within these fixed columns.

Step 3: Now that we have a set of rowSum values we need to take a consecutive set of these values, from top to bottom, that give the largest sum. If all the values were positive, we would take a sum of all the values and our rectangle would start from the first row till the end. But since we also have negative numbers, at one point even if we have positive numbers, there could be larger negative numbers that result in the total net sum becoming negative.

A simple solution in that case is to follow the Kadane algorithm, where we just reset the sum to be 0 and consider only the next upcoming rows untill the last row for our rectangle. We do this as we know that the sum so far cannot contribute positively to the maximum total sum of consecutive rowSum values.

Step 4: Finally, we get the maximum sum of the rectangle where the left border = leftColumn, right border = rightColumn, topBorder = latest restarted row/ first row and bottomBorder = lastRow that the relative maxSum was found in.

Special Case: In the case where all the row sum values are negative, simple return the smallest negative number as the sum and the rectangle only has the row of the number in it.

Step 5: In the end, all these rectangle sums are compared and we return the relative largest sum.

```
KanadeSum(rowSum[1..R])
maxSum = −inf
finish = −1
sum = 0
localStart = 0
for (i ← 1 : R)
  sum = sum + rowSum[i]
  if (sum < 0) then
        sum = 0
        localStart = i + 1
  else if (sum > maxSum)
        maxSum = sum
        start = localStart
        finish = i
if (finish ≠ −1) then
  return {maxSum, start, finish}
maxSum = rowSum[1]
start = finish = 1 for (i ← 2 : N)
  if (rowSum[i] > maxSum) then
        maxSum = rowSum[i] start = finish = 1
return {maxSum, start, finish}
```

```
MaximumRectangleSum(A[1..R][1..C])
maxRecSum = -inf
for (left ← 1 : C)
  temp[1..R] ← 0
  for (right ← left : C)
        for (i ← 1 : R)
              temp[i] ← temp[i] + A[i][right]
        result[] = KADANESUM(temp)
        recSum = result[1]
        startRow = result[2]
        finishRow = result[3]
        if (recSum > maxRecSum)then
              maxRecSum = recSum
              finalLeft = left
              finalRight = right
              finalTop =startRow
              finalBottom = finishRow
return maxRecSum
```

■

6. **Rod cutting:** The rod cutting problem assumes you have some rod of length n that you need to sell. The issue is that the market is illogical and rod price is not linearly proportional with rod length.

   Problem: You are given an integer x that represents the length of rod you have and an array A where i corresponds to a rod length and $A[i]$ corresponds to the price a rod of that length would fetch. You need to determine the maximum value you can fetch from the rod assuming you cut it optimally.

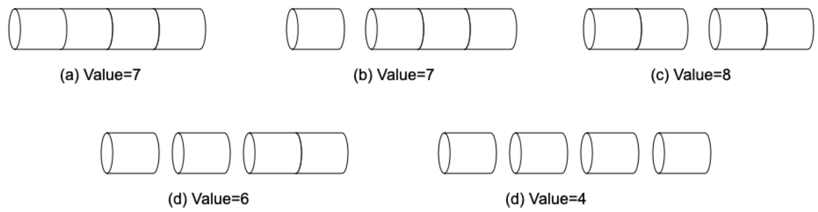
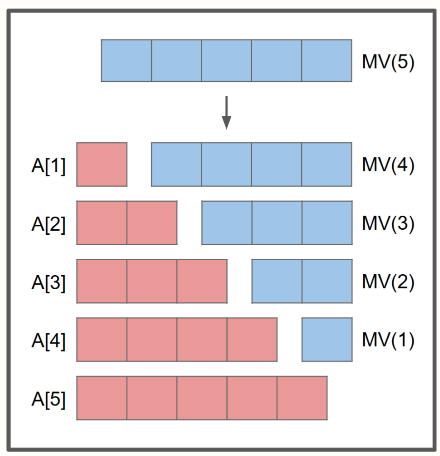


Figure 3. Example: $A = [1, 4, 6, 7]$ and $x = 4$, output should be **8**.

Solution: Suppose we decided to cut and sell a rod of length k out of the rod of length n. The maximum value you can get in this scenario would be $A[k]$ plus the maximum value you can get from a rod of length $n - k$.



However, we do not know if selling a rod of length k actually maximizes the total value. Therefore, to get the maximum total value for a rod of length n, we must try every $k \leq n$ and choose the k that gives the greatest value. With the observation, we can construct the following recurrence:

$$MV(n) = \max_{0 \leq i < n} \big(A[i] + MV(n - i)\big)$$

Where $MV(i)$ denotes the maximum value we can get from a rod of length i.

For a DP algorithm, we can memoize the values of MV in a one dimensional array $MV[1..n]$. Since we need the values of $MV[j]$ for all $j < i$ to compute $MV[i]$, we can start by filling out $MV[0]$ and proceed to the greater index. The pseudo-code of the algorithm is given below:

```
MV(n):
    MV[0] ← 0
    for i ← 1 to n:
        v ← 0
        for j ← 1 to i:
            if A[j] + MV[i − j] > v:
                v ← A[j] + MV[i − j]
        MV[i] ← v
    return MV[n]
```

Since we need to iterate through an array of size n, and each iteration takes $O(n)$ computation for computing the max, the runtime of the DP algorithm is $O(n^2)$.

■

7. In lecture we discussed the following two problems:

   - **Longest increasing subsequence (LIS)** - Given an array ($A[1..n]$) of n integers find the longest increasing subsequence.
   - **Longest common subsequence (LCS)** - Given two arrays ($A[1..n]$ and $B[1..n]$), what is the length of the longest subsequence present in both (for the sake of simplicity let's assume both arrays are of size n).

   Now I want the Longest Common Increasing Sub-sequence: given two arrays (A and B) each containing a sequence of n integers, what is the length of the longest subsequence that is present in both arrays.

Solution: Let us write LCIS(i, j) the length of the longest common increasing sequence of $A[1..i]$ and $B[1..j]$ that includes $B[j]$ for some $1 \le i, j \le n$. There are two scenarios to consider when computing LCIS(i, j): $A[i] \ne B[j]$ and $A[i] = B[j]$.

When $A[i] \ne B[j]$, $A[i]$ and $B[j]$ cannot be paired to be attached on the sequence which implies that $B[j]$ must be paired with one of the elements in $A[1..i-1]$. Therefore, in this case, LCIS(i, j)=LCIS$(i-1, j)$.

When $A[i] = B[j]$, $A[i]$ and $B[j]$ can be paired and attached to one of the common increasing sequences in $A[1..i-1]$, $B[1..j-1]$. However this is not possible for every sequence in $A[1..i-1]$, $B[1..j-1]$, since $A[i](= B[j])$ must be greater than the last element in the sequence to form an increasing sequence. Therefore, if we define $S(i, j) = \{k \mid 1 \le k < j, B[k] < A[i]\}$ the set of indices $k < j$ such that $B[k]$ is smaller than $A[i]$, then we have the following recurrence.

$$\mathrm{LCIS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \mathrm{LCIS}(i-1, j) & \text{if } i, j > 0 \text{ and } A[i] \ne B[j] \\ 1 + \max\limits_{k \in S(i,j)} \mathrm{LCIS}(i-1, k) & \text{if } i, j > 0 \text{ and } A[i] = B[j] \end{cases}$$

At a glance, we have n^2 subproblems, and each subproblem seems to have time complexity of $O(n)$, due to the max over $S(i, j)$. However, note that the max value does not have to be computed everytime we call LCIS. For any indices a, b, c such that $b < c$, we have

$$\max_{k \in S(a,b)} \mathrm{LCIS}(i-1, k) \le \max_{k \in S(a,c)} \mathrm{LCIS}(i-1, k)$$

Therefore, for each value of i, we can keep track of the maximum value we observed so far as we iterate through j, and directly access the value without recomputing the max. The psuedo-code of the algorithm is given below:

```
LCIS(i, j):
  for i ← 1 to n:
      m ← 0
      for j ← 1 to n:
          LCIS[i][j] → LCIS[i − 1][j]
          if A[i] > B[j] and LCIS[i][j] > m then m ← LCIS[i][j]
          if A[i] = B[j] then LCIS[i][j] ← m + 1
  return max_{1≤j≤n} LCIS[n][j]
```

Since we have n^2 subproblems, each with $O(1)$ time complexity, the overall time complexity of the algorithm is $O(n^2)$.

■