# Dynamic Programming

Josh Wilcox (jw14g24)

March 25, 2025

# Table of Contents

# Introduction

- Dynamic Programming -
  - Solve a given problem by combining solutions to smaller **subproblems**
- DP Applies when a problem has:
  - **Optimal Substructure**
    - Where there is a recurrence relation between the optimal solutions and its subproblems
  - **Overlapping Subproblems**
    - Otherwise use divide-and-conquer
- This can be solved by **memoization** or **bottom-up** solution
  - Finding recurrence is the hardest part

# Memoization and Bottom-p

- The fibonnaci system can easily be defined using recursive definition
- However **never ever ever** recurse on fibonacci because stoopid
- Memoization and bottom up can be used to remove the stoopidness of recursing through fibonacci

# Memoization on Fibonacci



- Stores the value of each subproblem to avoi recomputing the same subproblem multiple times
- Makes the time/space complexity $\Theta(n)$

**Algorithm** Memoized Fibonacci

```
1: create global array fibs of size n + 1
2: for i = 0 to n do
3:     fibs[i] ← −1
4: end for
5: fibs[0], fibs[1] ← 0, 1
6: function FB(n)
7:     if fibs[n] ≠ −1 then
8:         return fibs[n]
9:     else
10:        fibs[n] ← fibs[n − 1]+ fibs[n − 2]
11:        return fibs[n]
12:    end if
13: end function
```

# Bottom Up Fibonacci



- Constructs solutions from smaller to larger subproblems
- Also achieves $\Theta(n)$ time complexity

**Algorithm** Bottom Up Fibonacci

```
1: function FB(n)
2:     fibs[0], fibs[1] ← 0, 1
3:     for i = 2 to n do
4:         fibs[i] ← fibs[i − 1]+ fibs[i − 2]
5:     end for
6:     return fibs[n]
7: end function
```

# Independent Sets

- Given a graph $G = (V, E)$
  - An independent set is a subset of vertices $S \subseteq V$ such that for all $x, y \in S, \quad (x, y) \notin E$
- Usually we want $|S|$ to be maximum
- For the (linear) inependent set:
  - All nodes are in a straight line, so we just take every **other** nodes in the graph and adds them to the set

# Dynamic Programming Solution with Independent Sets



- Given a set of nodes $\{x_1, \ldots, x_n\}$ with associated weights $\{w_1, \ldots, w_n\}$
- We want to find the maximum weight independent set $S$ such that:

$$\sum_{x \in S} w_x \text{ is maximized}$$

- Let's denote $opt(j)$ as the maximum weight possible for nodes $\{x_1, \ldots, x_j\}$
- **Key insight**: For each node, we have two choices:
  - **Include** node $x_j$ in our solution
    - If we include $x_j$, we can't include $x_{j-1}$ (adjacent nodes)
    - So we add $w_j$ to the optimal solution for nodes $x_1$ through $x_{j-2}$
  - **Exclude** node $x_j$ from our solution
    - If we exclude $x_j$, the optimal solution is the same as for nodes $x_1$ through $x_{j-1}$
- This gives us the recurrence relation:

$$opt(j) = \max \begin{cases} w_j + opt(j-2) & (\text{include } x_j) \\ opt(j-1) & (\text{exclude } x_j) \end{cases}$$

- Base cases: $opt(0) = 0$ and $opt(1) = w_1$
- We choose whichever option gives the maximum value

We can use **Bottom Up** to make this more efficient - storing the lowest values to prevent unneeded recalculation

# Dynamic Programming Solution with Independent Sets - **Finding the solution**

- Once we have the **optimal value** we can **walk backwards** to get the actual solution (the nodes)
- If $opt(n) = opt(n-1)$ then we can conclude $x_n$ was not selected
- If $opt(n) \neq opt(n-1)$ then we can conclude that $x_n$ **was** selected and it follows that $x_{n-1}$ cannot be in the solution

# Interval Scheduling

- Greedy strategy of selecting intervals with the smallest finishing time first does not work when the intervals have weights
- We can solve this problem with DP - Here is the formula:
  - Let $I$ be a set of intervals where each interval $i$ starts at $e(i)$, ends at $f(i)$, and has a value $v(i)$
  - We want to find a subset of **non-overlapping** intervals $S \subseteq I$ such that we **maximize:**

$$V = \sum_{i \in S} v(i)$$

# DP Solution



- Let $I$ be the set of intervals we are dealing with
- Let $\mathcal{O}_n$ be the optimal solution for the first $n$ intervals
- Let $overlap(i_k)$ be the set of intervals overllaping with interval $k$ and denote the remaining set of intervals by:

$$I_k = I \; overlap(i_k)$$

$$opt(\mathcal{O}_n) = \max \begin{cases} v_n + opt(I_n) & i_n \in \mathcal{O}_n \\ opt(\mathcal{O}_{n-1}) & i_n \in \mathcal{O}_n \end{cases}$$

- Computing $overlap(i_k)$ implies we have to do $n-1$ overlap tests which leads to an $\Omega(n^2)$ agorithm
- We can optimize this computation:
  - Sort intervals by finish time
  - Define $p(j)$ as the largest index $i < j$ where interval $i$ is compatible with interval $j$
  - For each interval $j$, $p(j)$ can be computed in $O(\log n)$ time using binary search
  - This gives us the recurrence relation:

$$opt(j) = \max \begin{cases} v_j + opt(p(j)) & i_n \in \mathcal{O}_n \\ opt(j-1) & i_n \notin \mathcal{O}_n \end{cases}$$

  - Which can be solved in $O(n \log n)$ time

# DP Solution - **Bottom Up Value Solution**

**Algorithm** Bottom up solution for weighted interval scheduling

```
1: function OPT(v, p)
2:     opt[0] ← 0
3:     opt[1] ← v[1]
4:     for i = 2 to n do
5:         opt[i] ← max(opt[i − 1], v[i] + opt[p[i]])
6:     end for
7:     return opt
8: end function
```

- The algorithm fills the *opt* array bottom-up (from smaller to larger subproblems)
- $opt[i]$ represents the maximum value achievable using intervals 1 through $i$
- For each interval $i$, we decide whether to:
  - Skip it (take $opt[i-1]$), or
  - Include it (take $v[i] + opt[p[i]]$), where $p[i]$ is the last non-overlapping interval
- We take the maximum of these two choices
- Base cases: $opt[0] = 0$ (empty set) and $opt[1] = v[1]$ (just the first interval)
- Time complexity: $O(n)$ after preprocessing to compute $p$ values
- Space complexity: $O(n)$ for the *opt* array

# DP Solution - **Computing the solution**

**Algorithm** Computing optimal list of intervals for WIS

```
1: function OPT-SOL(opt, p)
2:     i ← n
3:     S ← ∅
4:     while i > 0 do
5:         if opt[i] > opt[i − 1] then
6:             S ← S ∪ {i}
7:             i ← p[i]
8:         else
9:             i ← i − 1
10:        end if
11:    end while
12:    return S
13: end function
```

- This algorithm reconstructs the set of intervals in the optimal solution
- Starting from the last interval ($i = n$), we work backwards:
  - If $opt[i] > opt[i-1]$, it means interval $i$ was included in the optimal solution
  - When we include interval $i$, we jump to $p[i]$ (the last compatible interval)
  - If $opt[i] = opt[i-1]$, it means interval $i$ was not included, so we move to $i-1$
- The set $S$ gradually collects all intervals in the optimal solution
- Time complexity: $O(n)$ in the worst case, as we process each interval at most once
- The algorithm terminates when we've examined all relevant intervals ($i = 0$)

# Longest Common Subsequence

- Leetcode #1143
- Given two strings $X$ and $Y$. Our goal is to find the longest subsequence contained in both $X$ and $Y$
  - E.g The common subsequence in `BDBDC` and `ADBC` is `DBC`
  - Notice how it is **different to substrings**
    - The longest common substring would be `DB`

# Dynamic Programming Solution



- Consider strings $X = x_1 \dots x_n$ and $Y = y_1 \dots y_m$
  - $n$ is the length of $X$
  - $m$ is the length of $Y$
- Denote prefix of length $k$ of both strings as:

$$X_k = x_1 \dots x_k$$

$$Y_k = y_1 \dots y_k$$

- Let $LCS(X, Y)$ be the length of the longest common subsequence between $X$ and $Y$

$$LCS(x, y) = LCS(X_n, Y_m)$$

# Dynamic Programming Solution - **The Recursion**

- If $x_n = y_m$, then the LCS is at least 1 therefore:

$$LCS(X_n, Y_m) = LCS(X_{n-1}, Y_{n-1}) + 1 \quad \text{if}$$

- If $x_n \neq y_m$, then $x_n$ and $y_m$ can't be in the solution so use DP programming:

$$LCS(X_n, Y_m) = \max \begin{cases} LCS(X_n, Y_{m-1}) \\ LCS(X_{n-1}, Y_m) \end{cases}$$

# Dynamic Programming Solution - **Bottom Up Algorithm**

**Algorithm** Longest Common Subsequence

```
1:  function LCS-DP(X, Y)
2:      n ← |X|                                          ▷ Get length of string X
3:      m ← |Y|                                          ▷ Get length of string Y
4:      Create 2D array OPT of size (n+1) × (m+1) ▷ Initialize our DP table with extra row/column for base cases
5:      for i = 0 to n do
6:          OPT[i][0] ← 0                                ▷ Base case: empty Y string means LCS length is 0
7:      end for
8:      for j = 0 to m do
9:          OPT[0][j] ← 0                                ▷ Base case: empty X string means LCS length is 0
10:     end for
11:     for i = 1 to n do
12:         for j = 1 to m do
13:             if X[i] = Y[j] then                      ▷ Characters match, extend the LCS
14:                 OPT[i][j] ← 1 + OPT[i − 1][j − 1]    ▷ Add 1 to LCS from previous characters
15:             else                                     ▷ Characters don't match
16:                 OPT[i][j] ← max(OPT[i − 1][j], OPT[i][j − 1])   ▷ Take best of excluding either character
17:             end if
18:         end for
19:     end for
20:     return OPT                                       ▷ Return the completed DP table
21: end function
```

- The algorithm builds a table ($OPT$) where $OPT[i][j]$ represents the length of the LCS between $X[1...i]$ and $Y[1...j]$
- We initialize the first row and column to 0 (base cases for empty strings)
- For each character position $(i, j)$:
  - If characters match ($X[i] = Y[j]$), we add 1 to the LCS from the previous positions
  - If characters don't match, we take the maximum LCS by either:
    - Excluding the current character from X ($OPT[i-1][j]$)
    - Excluding the current character from Y ($OPT[i][j-1]$)
- Time complexity: $O(nm)$ where $n$ and $m$ are the lengths of the strings
- Space complexity: $O(nm)$ for the DP table

# Knapsack

The Knapsack Problem

$$\text{Maximize: } \sum_i x_i \cdot v_i$$

$$\text{Subject To: } \sum_i x_i \cdot v_i \leq C$$

Can have other variants:

- Bounded Knapsack
  - $x_i \in \{0, 1, \ldots, b_i\}$ where $b_i$ is the maximum available quantity of item $i$
  - Each item has a specific upper limit on how many can be selected
  - Example: Limited inventory of different products
- Unbounded Knapsack
  - $x_i \in \mathbb{N}$ (non-negative integers)
  - Can select any number of each item (unlimited supply)
  - Example: Coin change problem - make change using unlimited coins of different denominations
- $0 - 1$ Knapsack
  - $x_i \in \{0, 1\}$
  - Item can be used zero or one time each

# DP Solution for 0-1



- Let $S_n$ be the optimal solution when the problem contains $n$ items with capacity $C$
- For item $i_n$ there are two possibilities:
  - $i_n \notin S_n$
    - $n-1$ remaining items to look through
    - Remaining capacity is still $C$
  - $i_n \in S_n$
    - $n-1$ remaining items to look through
    - Remaining capacity is now $C - w_n$ as we added that item to the knapsack

$$S(n, C) = \max \begin{cases} v_n + S(n-1, C - w_n) \\ S(n-1, C) \end{cases}$$

Base Cases:

- $S(k, 0) = 0 \quad \forall k \in \mathbb{R}$
- If $w_n > C, \quad S(n, C) = S(n-1, C)$

# Bottom-Up Pseudocode



**Algorithm** Bottom up solution for 0-1 Knapsack

```
1:  function KNAPSACK(w, v, C)
2:      for i = 1 to n do
3:          OPT[i][0] ← 0
4:      end for
5:      for j = 1 to C do
6:          OPT[0][j] ← 0
7:      end for
8:      for j = 1 to C do
9:          for i = 1 to n do
10:             if j ≥ w[i] then
11:                 OPT[i][j] ← max(OPT[i − 1][j], v[i] + OPT[i − 1][j − w[i]])
12:             else
13:                 OPT[i][j] ← OPT[i − 1][j]
14:             end if
15:         end for
16:     end for
17:     return OPT
18: end function
```

- The algorithm builds a table ($OPT$) where $OPT[i][j]$ represents the maximum value achievable using items 1 through $i$ with capacity $j$
- We initialize the first row and column to 0 (base cases for empty knapsack or zero capacity)
- For each item $i$ and each possible capacity $j$:
  - If the current item's weight fits ($w[i] \leq j$), we consider two options:
    - Skip the item: $OPT[i-1][j]$
    - Take the item: $v[i] + OPT[i-1][j-w[i]]$
  - If the item doesn't fit ($w[i] > j$), we can only skip it
- Time complexity: $O(nC)$ where $n$ is the number of items and $C$ is the capacity
- Space complexity: $O(nC)$ for the DP table

| $n \backslash w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 0 | 0 | 0 | 2 | 2 | 2 | (4) | 4 | 4 | 6 | 6 | 6 | 6 | 6 |
| 3 | 0 | 0 | 0 | 2 | 2 | 2 | 4 | 5 | 5 | 6 | 7 | 7 | 7 | (9) |
| 4 | 0 | 0 | 0 | 2 | 2 | 2 | 4 | 5 | 6 | 6 | 7 | 8 | 8 | 9 |

# Finding Solution from Matrix



| $n \backslash w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 0 | 0 | 0 | 2 | 2 | 2 | (4) | 4 | 4 | 6 | 6 | 6 | 6 | 6 |
| 3 | 0 | 0 | 0 | 2 | 2 | 2 | 4 | 5 | 5 | 6 | 7 | 7 | 7 | (9) |
| 4 | 0 | 0 | 0 | 2 | 2 | 2 | 4 | 5 | 6 | 6 | 7 | 8 | 8 | 9 |

**Algorithm** Bottom up solution for 0-1 Knapsack

```
1: function KNAPSACK(OPT, w, C, n)
2:     w' ← C
3:     L ← []
4:     i ← n
5:     while i > 0 ∧ w' > 0 do
6:         if OPT[i][w'] > OPT[i − 1][w'] then
7:             L.push_front(i)
8:             w' ← w' − w[i]
9:         end if
10:        i ← i − 1
11:    end while
12:    return L
13: end function
```

- The algorithm reconstructs which items were selected in the optimal solution by working backwards
- Starting at cell $OPT[n][C]$ (bottom-right of matrix), which represents the optimal value for all items and full capacity
- For each position $(i, w')$ in the table:
  - If $OPT[i][w'] > OPT[i-1][w']$, it means item $i$ was included in the optimal solution
  - When an item is included, we:
    - Add the item to our solution list
    - Reduce the remaining capacity by the item's weight
    - Move diagonally up-left in the matrix to position $(i-1, w'-w[i])$
  - If $OPT[i][w'] = OPT[i-1][w']$, it means item $i$ was not included, so we simply move up in the matrix
- We continue this process until we either:
  - Run out of items ($i = 0$), or
  - Run out of capacity ($w' = 0$)
- The final list $L$ contains all items in the optimal knapsack solution

# Rod Cutting

**Problem Overview:**

- A steel company wants to maximize revenue by cutting a rod into smaller pieces.
- Each integer length $i$ has an associated price $p[i]$.
- Given a rod of length $n$, determine the optimal set of cuts to maximize total revenue.

**Hikmations**

$$r(c_{n-2}, n) = \max \begin{cases} r(c_{n-3}, n) \\ r(2) + r(n-2) \end{cases}$$

$$r(c_{n-2}, n) = \max \begin{cases} r(c_{n-3}, n) \\ r(2) + r(n-2) \end{cases}$$

**Derivation of the Recursive Solution:**

- Consider making a cut at length $k$ (for every $1 \leq k \leq n$).
- Cutting the rod at $k$ yields a piece of length $k$ which earns revenue $p(k)$, and leaves a remaining rod of length $n-k$.
- The problem then reduces to finding the optimal revenue for the remaining rod; that is, $opt(n-k)$.
- Evaluating all possible first cuts gives the recurrence:

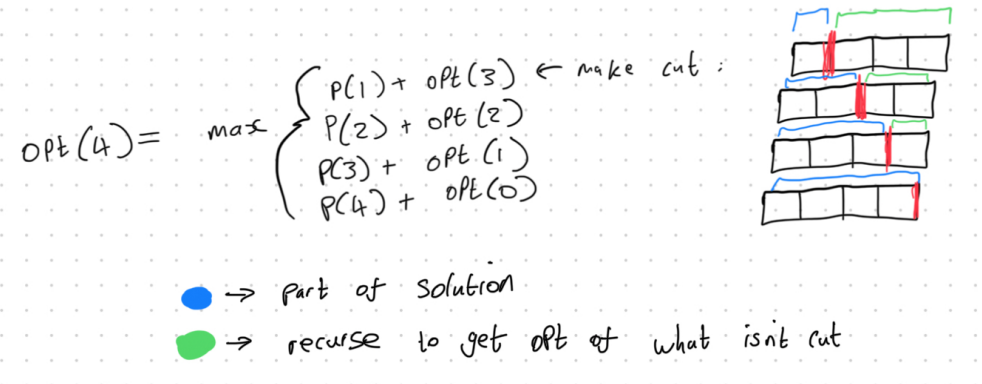
$$opt(n) = \max_{1 \leq k \leq n} \{p(k) + opt(n-k)\}$$

- The base case is defined as:

$$opt(0) = 0$$

  meaning that a rod of length 0 produces no revenue.

**Visualisation:**



# Rod Cutting Pseudocode

### Recursive Algorithm

```
1  int cutRod(p, n){
2      if (n==0){
3          return 0
4      }
5      r = - infinity
6      for (i = 1; i <= n; i++){
7          r = max(r, p[i] + cutRod(p, n-i))
8      }
9      return r
10 }
11
```

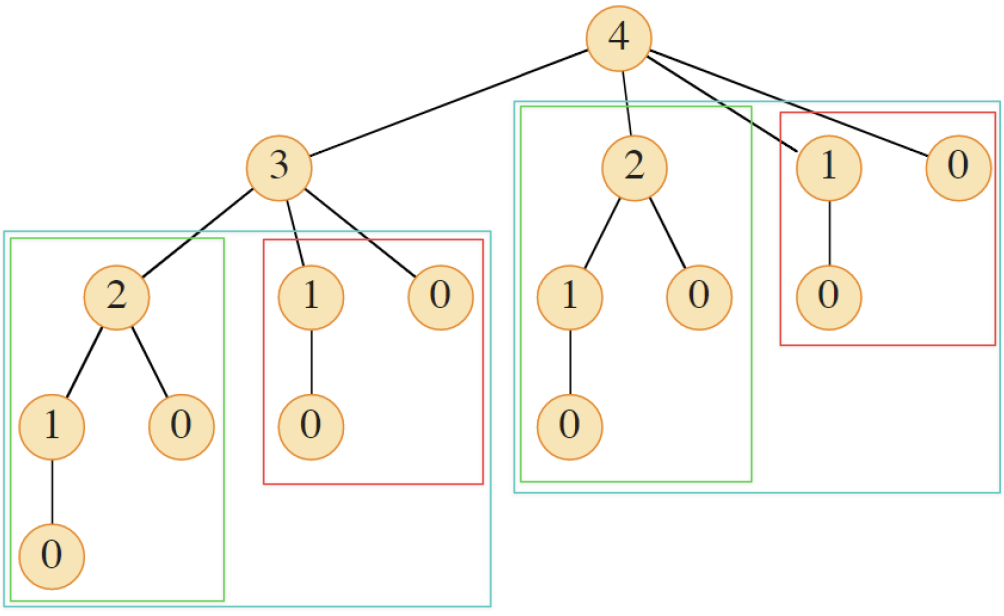
### Complexity

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=0}^{n-1} T(i) \\
&= \left(1 + \sum_{i=0}^{n-2} T(i)\right) + T(n-1) \\
&= T(n-1) + T(n-1) \\
&= 2T(n-1)
\end{aligned}
$$

$$\dots$$

$$T(n) = 2^n$$

- This is a bad complexity due to the repeated computation of subproblems



# Memoization



**Algorithm** Memoized algorithm for rod cutting

```
1: Initialize array r[1...n]                                   ▷ Create array to store computed results
2: for i ← 1 to n do
3:     r[i] ← −∞                                               ▷ Initialize all values to negative infinity (indicating not computed yet)
4: end for
5: function MEMO-CUT-ROD(p, n)
6:     if r[n] ≥ 0 then                                        ▷ Check if we already computed this subproblem
7:         return r[n]                                         ▷ Return the cached result (memoization)
8:     end if
9:     if n = 0 then                                           ▷ Base case: rod of length 0
10:        return 0                                            ▷ No revenue from empty rod
11:    end if
12:    m ← −∞                                                  ▷ Initialize maximum revenue to negative infinity
13:    for i ← 1 to n do                                       ▷ Try each possible first cut position
14:        m ← max(m, p[i] + MEMO-CUT-ROD(p, n − i))           ▷ Take maximum of current best and cutting at position i
15:    end for
16:    r[n] ← m                                                ▷ Store result in memoization table
17:    return m                                                ▷ Return the maximum revenue
18: end function
```

- This algorithm improves the time complexity from $O(2^n)$ to $O(n^2)$
- The array $r$ stores previously computed results to avoid redundant calculations
- Key improvements:
  - Each subproblem is solved exactly once
  - Results are stored and reused when needed
  - The recursive calls only happen for subproblems we haven't yet solved
- Space complexity is $O(n)$ for the memoization array
- This is a top-down approach to dynamic programming - we still use recursion but avoid redundant calculations

# Bottom Up for Rod Cutting



**Algorithm** Bottom up algorithm for rod cutting

```
1: function BOTTOM-UP-CUT-ROD(p, n)
2:     r[0] ← 0                          ▷ Base case: rod of length 0 has no revenue
3:     for j ← 1 to n do                 ▷ Build solutions for rods of increasing length
4:         m ← −∞                        ▷ Initialize maximum revenue to negative infinity
5:         for i ← 1 to j do             ▷ Try each possible first cut position
6:             m ← max(m, p[i] + r[j − i])   ▷ Take maximum of current best and cutting at position i
7:         end for
8:         r[j] ← m                      ▷ Store the optimal solution for length j
9:     end for
10:    return r[n]                       ▷ Return the maximum revenue for the full rod
11: end function
```

- This bottom-up approach builds solutions iteratively from smaller to larger subproblems
- Key advantages:
  - Eliminates recursion overhead and stack space requirements
  - Solves each subproblem exactly once in a systematic order
  - Typically more efficient than the memoized approach in practice
- The outer loop considers rods of length 1 through n
- The inner loop tries all possible first cuts for each length
- Time complexity: $O(n^2)$ - we have two nested loops, each running at most n iterations
- Space complexity: $O(n)$ for the array r storing optimal solutions
- This approach guarantees we have all necessary subproblem solutions before they're needed

# Maximum Subarray Sum

### Defining The Problem

- Given Array (with at least one negative number):

$$A_0, A_1, \ldots A_{n-1}$$

- Find the largest **subarray** (contiguous chunk of the array) whose sum of elements are maximum
- I.e find $\max S_{ij}$ where:

$$S_{ij} = \sum_{k=i}^{j \geq i} A_k$$

### Finding the Solution

- Define:

$$M_j = \max_i S_{ij}$$

- Compute $M_j$ for all $j$ and find the maximum such that:

$$\max_{ij} S_{ij} = \max_j M_j$$

- *In other words*: For each ending position in the array, find the best possible starting position, then take the best of all these combinations

$$M_j = \max \begin{cases} A[1] + A[2] + A[3] + \cdots + A[k] + \cdots + A[j] \\ A[2] + A[3] + \cdots + A[k] + \cdots + A[j] \\ A[3] + \cdots + A[k] + \cdots + A[j] \\ \quad \ldots \\ A[k] + \cdots + A[j] \\ \quad \ldots \\ A[j] \end{cases}$$

- The Maximum of $M_{j+1}$ can either be $M_j + A[j+1]$ or the value $A[j+1]$ itself
  - If $M_j + A[j+1] > A[j+1]$, then extending the previous best subarray is optimal
  - If $A[j+1] > M_j + A[j+1]$, then starting a new subarray at position $j+1$ is optimal
  - This leads to the recurrence relation: $M_{j+1} = \max(M_j + A[j+1], A[j+1])$
  - Intuitively: if adding the current element improves our sum, keep extending; otherwise, start fresh
- Therefore:

$$M_j = \max \begin{cases} A[j] + M_{j-1} \\ A[j] \end{cases}$$

- The solution is the maximum value of $M_j$ we can find

# DP Pseudocode



**Algorithm** Maximum Subarray Sum

```
1:  function MaxSubarraySum(A)
2:      M[0] ← 0                                   ▷ Initialize M[0] to 0
3:      for i = 1 to n do                          ▷ Iterate through the array
4:          M[i] ← max(A[i], A[i] + M[i − 1])      ▷ Either start new subarray or extend previous one
5:      end for
6:      m ← 0                                      ▷ Initialize maximum sum to 0
7:      for i = 1 to n do                          ▷ Find the maximum value in M array
8:          if M[i] > m then                       ▷ Update if we find a larger sum
9:              m ← M[i]                           ▷ Update the maximum sum
10:             idx ← i                            ▷ Keep track of ending position
11:         end if
12:     end for
13:     return m, idx                              ▷ Return the maximum sum and its ending position
14: end function
```

# Subset Sum

- Given array $A$ of $n$ elements and an integer value $S$
- We need to find if there is a **subset** of $A$ whose sum is exactly $S$
- Starting with the last element, there are two choices:
  - Either the last element is in the sum:
    - Find subset of first $n-1$ elements whose sum is $S - A[n]$
  - Else it is not in the subset
    - Find subset of first $n-1$ elements whose sum is $S$

**Algorithm** Subset Sum

```
1: function SubsetSum(A, n, S)
2:     if S = 0 then                          ▷ First base case: target sum is 0
3:         return true                        ▷ Empty set has sum 0, solution found
4:     end if
5:     if n = 0 then                          ▷ Second base case: no elements left to consider
6:         return false                       ▷ Can't form the required sum with no elements
7:     end if
8:     return SubsetSum(A, n − 1, S) ∨ SubsetSum(A, n − 1, S − A[n])   ▷ Try both excluding and including the current element
9: end function
```

- **Understanding the algorithm:**
  - This is a recursive solution that explores all possible subsets
  - The algorithm makes a binary decision for each element: include it or exclude it
  - Time complexity: $O(2^n)$ due to exploring all possible subsets
- **If statement logic:**
  - **First if statement** ($S = 0$):
    - When we've reduced our target sum to exactly 0, we've found a valid subset
    - This happens when the elements we've chosen to include sum up exactly to our original target
    - Returns `true` immediately as a solution exists (empty set for $S = 0$)
  - **Second if statement** ($n = 0$):
    - When we've considered all elements but haven't reached a sum of 0
    - If we've run out of elements and $S \neq 0$, it's impossible to form the sum
    - Returns `false` as no solution is possible with the given constraints
  - **Recursive case** (logical OR of two possibilities):
    - `SubsetSum(A, n-1, S)`: Excludes the current element from the subset
    - `SubsetSum(A, n-1, S - A[n])`: Includes the current element in the subset
    - If either recursive call returns `true`, a valid solution exists
- **Optimization opportunities:**
  - This recursive solution can be optimized using dynamic programming
  - A 2D array can store results of subproblems to avoid redundant calculations
  - The DP solution would have $O(n \times S)$ time complexity

# Bottom Up Solution



```
function SubsetSum(A,S){
    // Initialize base case: empty set has sum 0
    for (i=0; i <=n; i++){
        opt[i][0] = true;
    }
    // Initialize base case: no elements can't form non-zero sum
    for (j=1; j <= S; j++){
        opt[0][j] = false;
    }

    // Fill DP table bottom-up
    for (i=1; i <= n; i++){
        for (j=1; j <= S; j++){
            if (j < A[i]) {
                // If current element is larger than current sum,
                // we can only exclude it
                opt[i][j] = opt[i-1][j];
            } else {
                // Either exclude current element OR include it
                opt[i][j] = opt[i-1][j] || opt[i-1][j-A[i]];
            }
        }
    }

    return opt[n][S]; // Return true if subset exists, false otherwise
}
```

- This bottom-up dynamic programming approach solves the Subset Sum problem efficiently
- The 2D array `opt[i][j]` represents whether a subset of the first `i` elements can sum to `j`
- Key aspects of the algorithm:
  - We initialize two base cases:
    - An empty set can always form a sum of 0 (`opt[i][0] = true`)
    - No elements can never form a non-zero sum (`opt[0][j] = false`)
  - For each element and possible sum value, we consider:
    - Can we form sum `j` without using element `i`? (`opt[i-1][j]`)
    - Can we form sum `j-A[i]` without element `i`, then add `A[i]`? (`opt[i-1][j-A[i]]`)
  - If either option is possible, then we can form sum `j` using the first `i` elements
- Time complexity: $O(n \times S)$ where $n$ is the number of elements and $S$ is the target sum
- Space complexity: $O(n \times S)$ for the DP table
- This algorithm is pseudopolynomial - polynomial in $n$ and $S$, but $S$ could be exponential in the input size

# Finding the Solution



**Algorithm** Finding the elements in the Subset Sum solution

```
1:  function SSSolution(A, S, opt)
2:      i, val ← n, S                          ▷ Start at bottom-right of matrix
3:      sol ← ∅                                ▷ Initialize empty solution set
4:      while i ≠ 0 ∧ val ≠ 0 do               ▷ Continue until we reach top row or zero sum
5:          if val − A[i] ≥ 0 ∧ opt[i − 1][val − A[i]] then   ▷ Check if current element is part of solution
6:              sol ← sol ∪ {A[i]}             ▷ Add current element to solution
7:              val ← val − A[i]               ▷ Reduce remaining sum by current element's value
8:          end if
9:          i ← i − 1                          ▷ Move to previous element regardless
10:     end while
11:     return sol                             ▷ Return the set of elements forming the subset
12: end function
```

- This algorithm traces back through the DP table to find the specific elements that form the subset sum
- Starting from the bottom-right of the matrix ($opt[n][S]$), we work backwards to identify which elements were included
- For each position in the matrix, we check:
  - If the current element $A[i]$ is part of the solution: This happens when excluding the current element still allows forming the sum $val - A[i]$ with the previous elements
  - If it is, we add the element to our solution set and adjust the remaining sum value
- The algorithm works by following the decisions made during the DP table construction:
  - If we included element $A[i]$ in forming sum $val$, then $opt[i-1][val - A[i]]$ must be true
  - When this condition is true, we know $A[i]$ is part of the solution
- We always decrement $i$ to move to the previous element, regardless of whether the current element was included
- The algorithm terminates when either:
  - We've processed all elements ($i = 0$), or
  - We've found all elements needed to form the sum ($val = 0$)
- The final set $sol$ contains exactly the elements that sum to $S$

# Bellman-Ford Single Source Shortest Path

- Bellman-Ford finds the shortest path from a source vertex to all other vertices in a weighted graph
- Let $d[v, i]$ be the shortest path from source $s$ to vertex $v$ using at most $i$ edges
- Base case: $d[s, 0] = 0$ and $d[v, 0] = \infty$ for all $v \neq s$

$$d[v, i+1] = \min \begin{cases} d[v, i] & \text{(don't use an additional edge)} \\ \min_{(u,v) \in E} \{d[u, i] + w(u, v)\} & \text{(use one more edge)} \end{cases}$$

# DP Algorithm



**Algorithm** Bellman-Ford Algorithm

```
1: function BELLMAN-FORD(V, inDeg)
2:     for all v ∈ V do
3:         d[0][v] ← ∞                                  ▷ Initialize distances to infinity
4:     end for
5:     d[0][s] ← 0                                      ▷ Distance from source to itself is 0
6:     for i ← 1 to |V| do
7:         for all v ∈ V do
8:             t ← min_{u∈inDeg(v)}(d[i − 1][u] + w(u, v))   ▷ Find minimum distance through incoming edges
9:             d[i][v] ← min(d[i − 1][v], t)            ▷ Keep best of previous or new distance
10:        end for
11:    end for
12:    return d                                         ▷ Return distance matrix
13: end function
```

- The algorithm builds a 2D table where $d[i][v]$ represents the shortest path from source $s$ to vertex $v$ using at most $i$ edges
- Key aspects of the algorithm:
  - We initialize distances: 0 for the source, infinity for all other vertices
  - For each possible path length $i$ (from 1 to —V—):
    - For each vertex, we consider all incoming edges
    - We update the distance if we find a shorter path through any incoming edge
    - We also consider keeping the previous best distance (using fewer edges)
  - After —V—-1 iterations, we've found all shortest paths (unless there are negative cycles)
  - The —V—th iteration can be used to detect negative cycles
- Time complexity: $O(|V| \times |E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges
- Space complexity: $O(|V|^2)$ for the distance matrix
- This DP approach systematically considers paths of increasing length until finding the optimal solution

# Floyd-Warshall

- Floyds algorithm finds the shortest path between **all pairs** of vertices in a weighted graph
- It uses the properties of **intermediate vertices of paths**
- Given a simple path $p = (v_1, \dots v_k)$ in $G = (V, E, w)$ where V $= v_1, \dots, v_n$:
  - An **intermediate** vertex is any vertex of $p$ other than $v_1$ and $v_k$
- Denote by $\Pi_{ij}^k$ the set of paths from $v_i$ to $v_j$ whose intermediate vertices are drawn from $\{v_1, \dots, v_k\}$
- Let $\pi \in \Pi_{ij}^k$ be a path with the **smallest weight**
- There are two cases:
  - Either $v_k \in \pi$
    - $\pi$ can be decomposed into two (shortest paths) $\pi_1$ and $\pi_2$ where:

$$\pi_1 \in \Pi_{ik}^{k-1} \quad \text{and} \quad \pi_1 \in \Pi_{kj}^{k-1}$$

    - for any shortest path $v_i, \dots, v_k, \dots, v_l, \dots, v_j$ the subpath $v_k, \dots, v_l$ is a shortest path from $v_k$ to $v_l$
  - Or $v_k \notin \pi$ therefore:

$$\pi \in \Pi_{kj}^{k-1}$$

**Recursion**

- Let $d_{ij}^k$ be the weight of the shortest path from $v_i$ to $v_j$ whose intermediate vertices are drawn from $\{v_1, \dots, v_k\}$

$$d_{ij}^k = \min \begin{cases} w(i,j) \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \end{cases}$$

# Bottom Up Algorithm



**Algorithm** Floyd-Warshall Algorithm

```
1: function FLOYD-WARSHALL(G)
2:     Initialize d where d[i][j] ← w(i,j) if (i,j) ∈ E, ∞ if i ≠ j, and 0 if i = j   ▷ Initialize distances using edge weights
3:     for k ← 1 to |V| do                                   ▷ Consider each vertex as an intermediate vertex
4:         for i ← 1 to |V| do                               ▷ Consider each possible starting vertex
5:             for j ← 1 to |V| do                           ▷ Consider each possible ending vertex
6:                 d[i][j] ← min(d[i][j], d[i][k] + d[k][j])   ▷ Update if path through k is shorter
7:             end for
8:         end for
9:     end for
10:    return d                                              ▷ Return the matrix of shortest path distances
11: end function
```

- The Floyd-Warshall algorithm builds a distance matrix where $d[i][j]$ represents the shortest path distance from vertex $i$ to vertex $j$
- Key aspects of the algorithm:
  - We initialize the distance matrix with direct edge weights, $\infty$ for non-adjacent vertices, and 0 for self-loops
  - The algorithm considers vertices in order from 1 to $|V|$ as potential intermediate vertices
  - For each triplet of vertices $(i, k, j)$, we check if going from $i$ to $j$ through $k$ gives a shorter path
  - If so, we update the distance $d[i][j]$ to use this shorter path
- The algorithm systematically relaxes distances by considering all possible intermediate vertices
- Time complexity: $O(|V|^3)$ due to the three nested loops
- Space complexity: $O(|V|^2)$ for the distance matrix
- After the algorithm completes, $d[i][j]$ holds the shortest path distance from $i$ to $j$ using any vertices in the graph as intermediates
- If there are negative cycles, they can be detected by checking if any $d[i][i] < 0$ after the algorithm completes