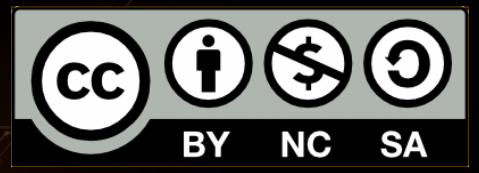
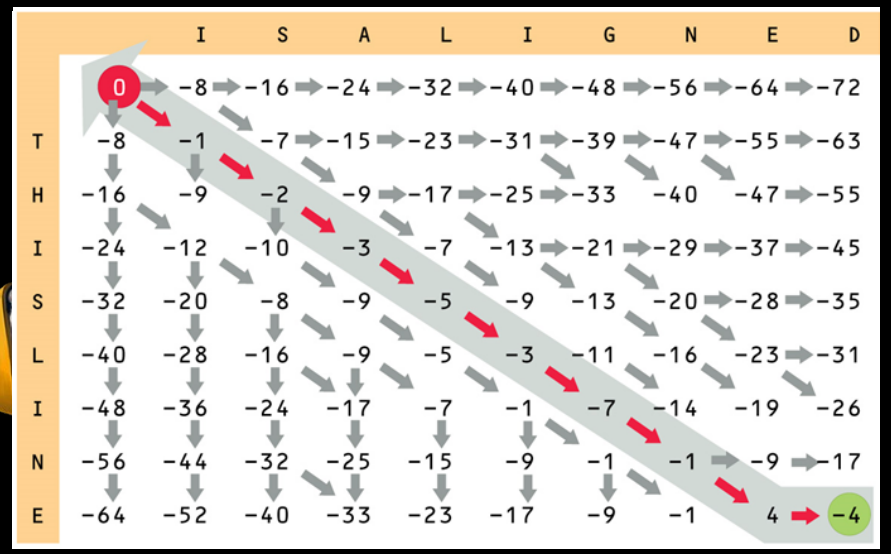
Dynamic Programming

Divide-and-Conquer, Dynamic Programming, and Memoization





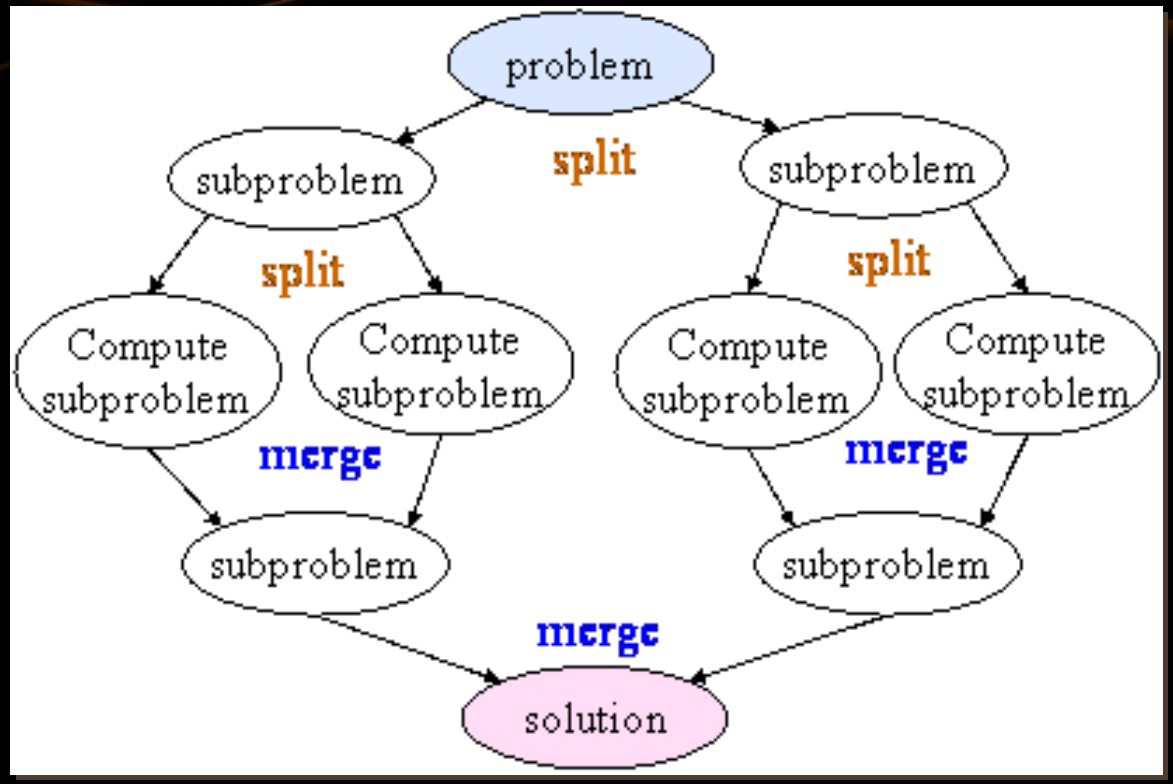
SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents



Divide-and-Conquer

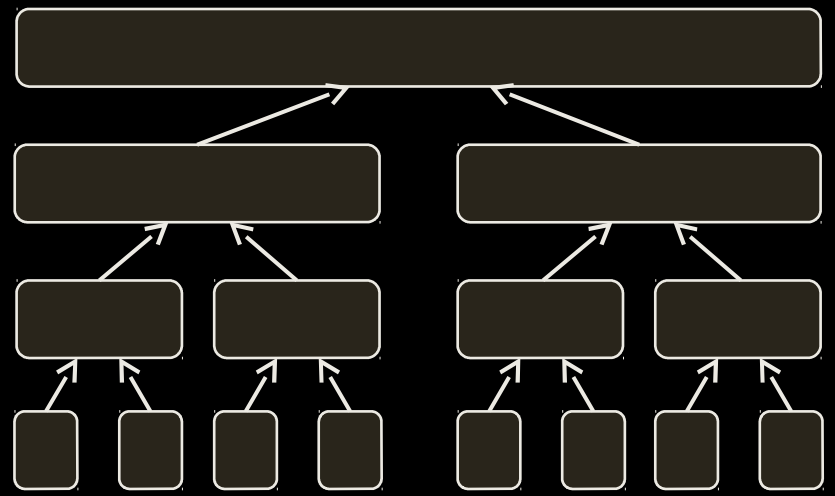
Divide-and-Conquer

- Divide-and-conquer programming technique divides the problem into disjoint sub-problems, solves them and combines the results
  - Examples: MergeSort, QuickSort, Map-Reduce algorithms
- Steps in divide-and-conquer
  - Divide: divide the problem into two or more disjoint sub-problems
    - Sub-problems are disjoint (not intersecting)
  - Conquer: conquer recursively to solve the subproblems
  - Combine: take the solutions to the subproblems and

Divide-and-Conquer Example

- MergeSort

```
void MergeSort(int[] arr, int left, int
right)
{
  if (right > left)
  {
    int mid = (right + left) / 2;
    MergeSort(arr, left, mid);
    MergeSort(arr, (mid+1), right);
    Merge(arr, left, (mid+1), right);
  }
}
```

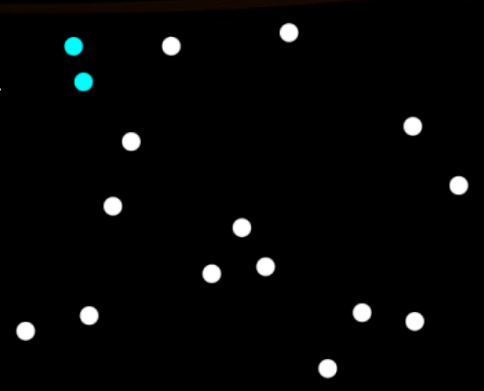


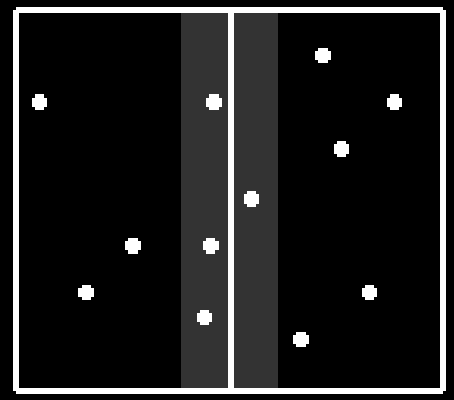
- The sub-problems are independent, all different

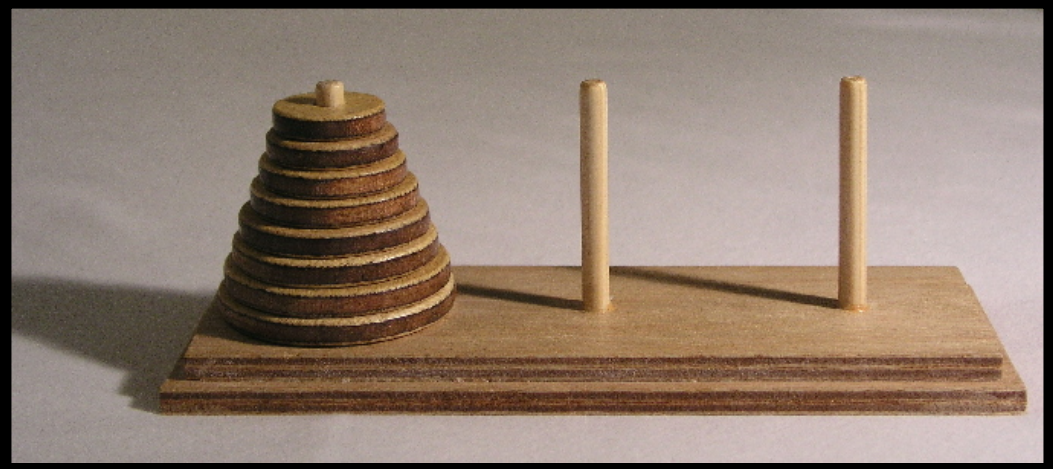
Divide-and-Conquer Algorithms

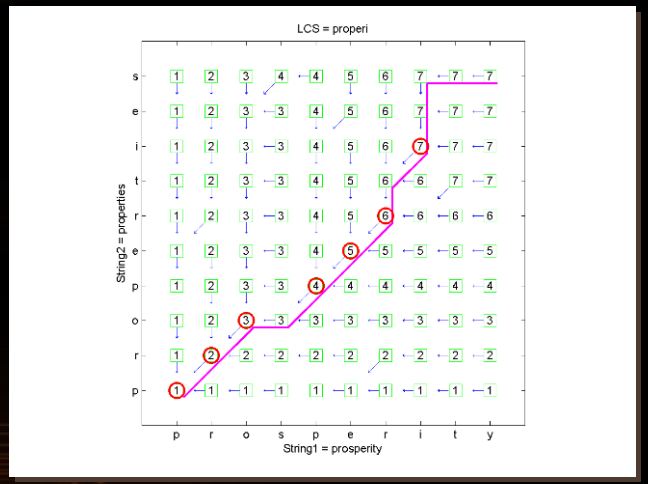
SOFTWARE UNIVERSITY FOUNDATION

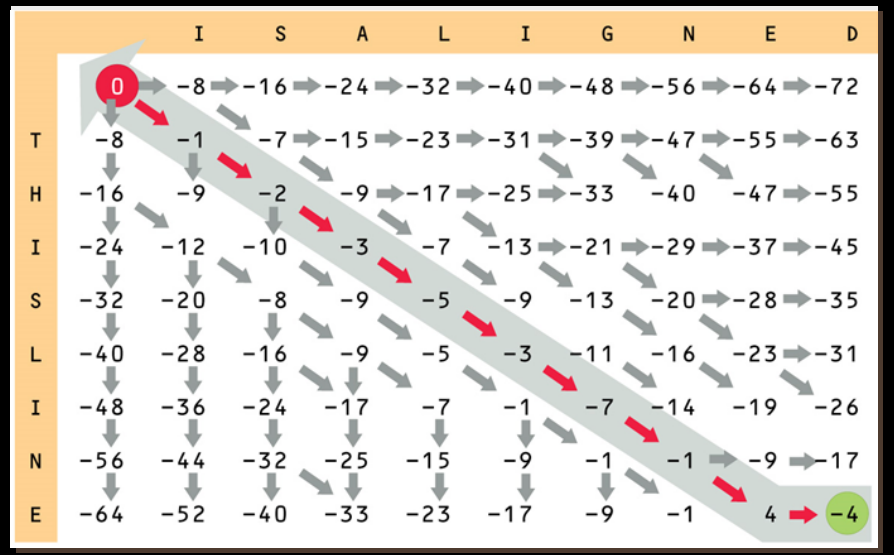
- Binary search
  - Closest pair in 2D geometry
- QuickSort
- Merging sorted arrays
  - MergeSort
- Finding majorant
- Tower of Hanoi
- Fast matrix multiplication

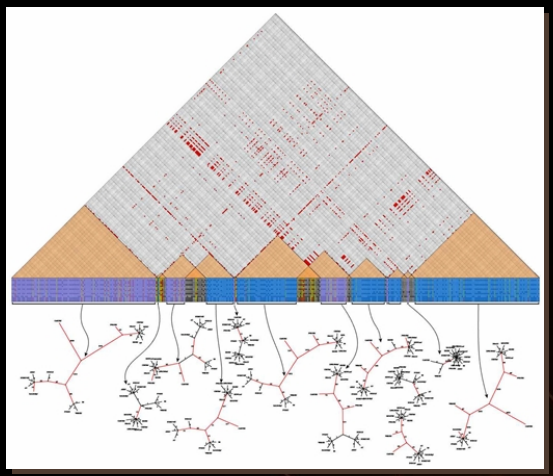












Dynamic Programming

Concepts

Dynamic Programming

- How dynamic programming (DP) works?
  - Solve problems by breaking them into smaller problems
  - Store partial solutions of the smaller problems
  - Combine smaller problem to solve the bigger problem
- Steps to design a DP algorithm:
  - Find an optimal substructure (problem solved by sub-problems)
  - Recursively define the value of an optimal solution
  - Compute the value bottom-up or top-down
  - Construct an optimal solution (if needed)

Characteristics of DP

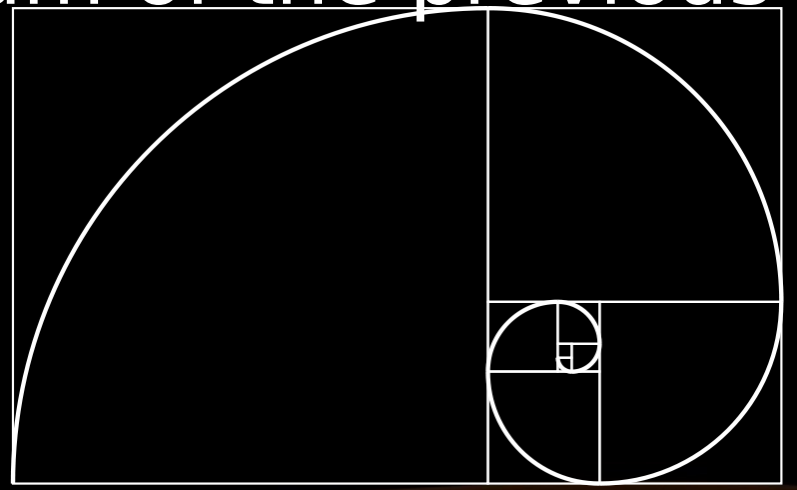
- Simple sub-problems
  - Break the original problem to smaller sub-problems
    - That have the same structure
- Optimal substructure of the problems
  - The optimal solution to the problem contains within optimal solutions to its sub-problems
- Overlapping sub-problems
  - Solution to the same sub-problem could be needed multiple times

Dynamic Programming vs. Divide-and-Conquer

- Use divide-and-conquer for non-overlapping sub-problems
  - Any sub-problem should be solved once only
  - Example: recursive factorial
- Use dynamic programming (DP) for overlapping sub-problems
  - Solve each sub-problem once store its answer in a table
  - Thus avoid solving the same sub-problems multiple times

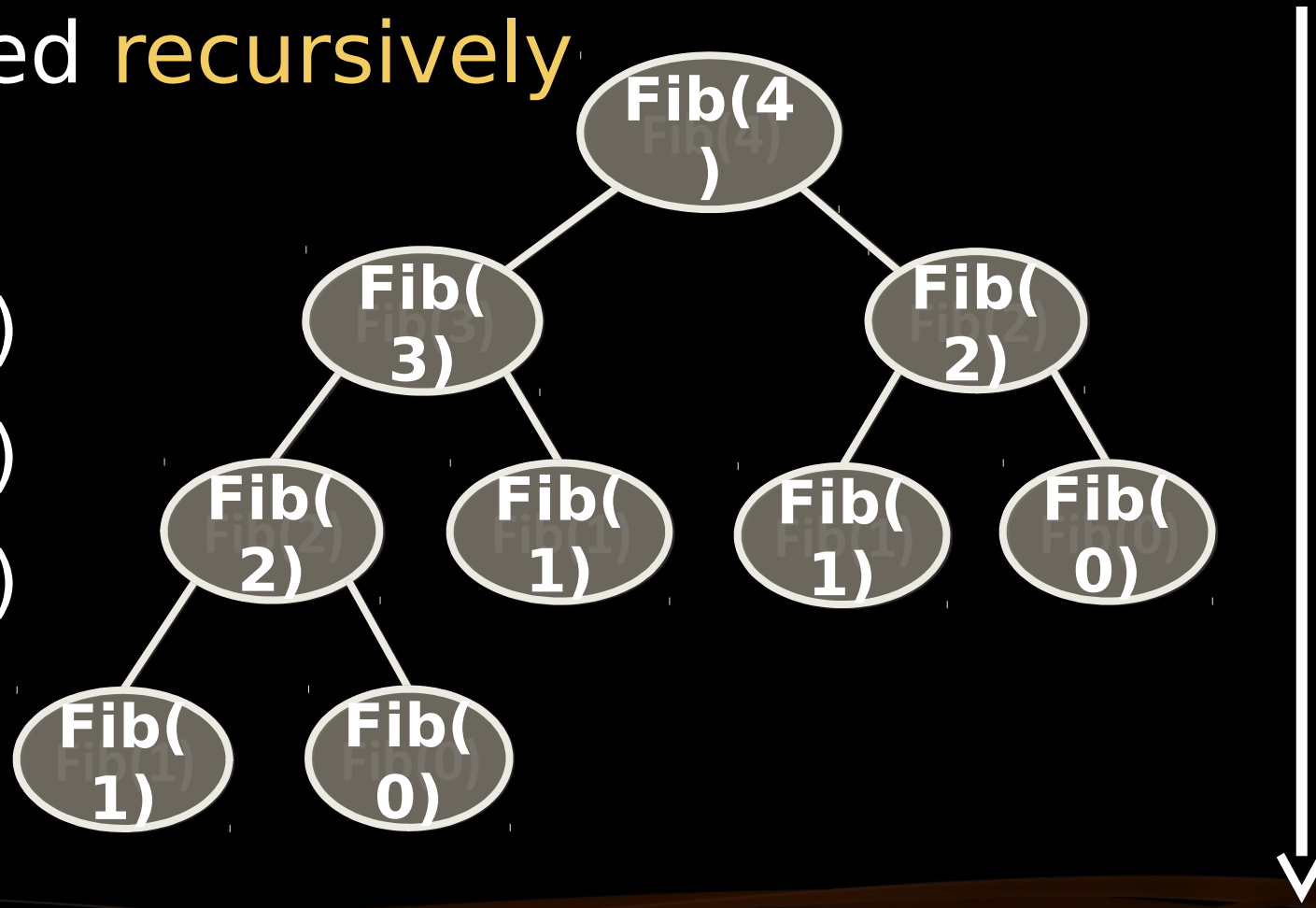
Example: Fibonacci Sequence

- The Fibonacci sequence holds the following integers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
  - The first two numbers are 0 and 1
  - Each subsequent number is the sum of the previous two numbers
- Recursive mathematical formula:
  - $F_0 = 0, F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$



Dynamic Programming: Top-Down

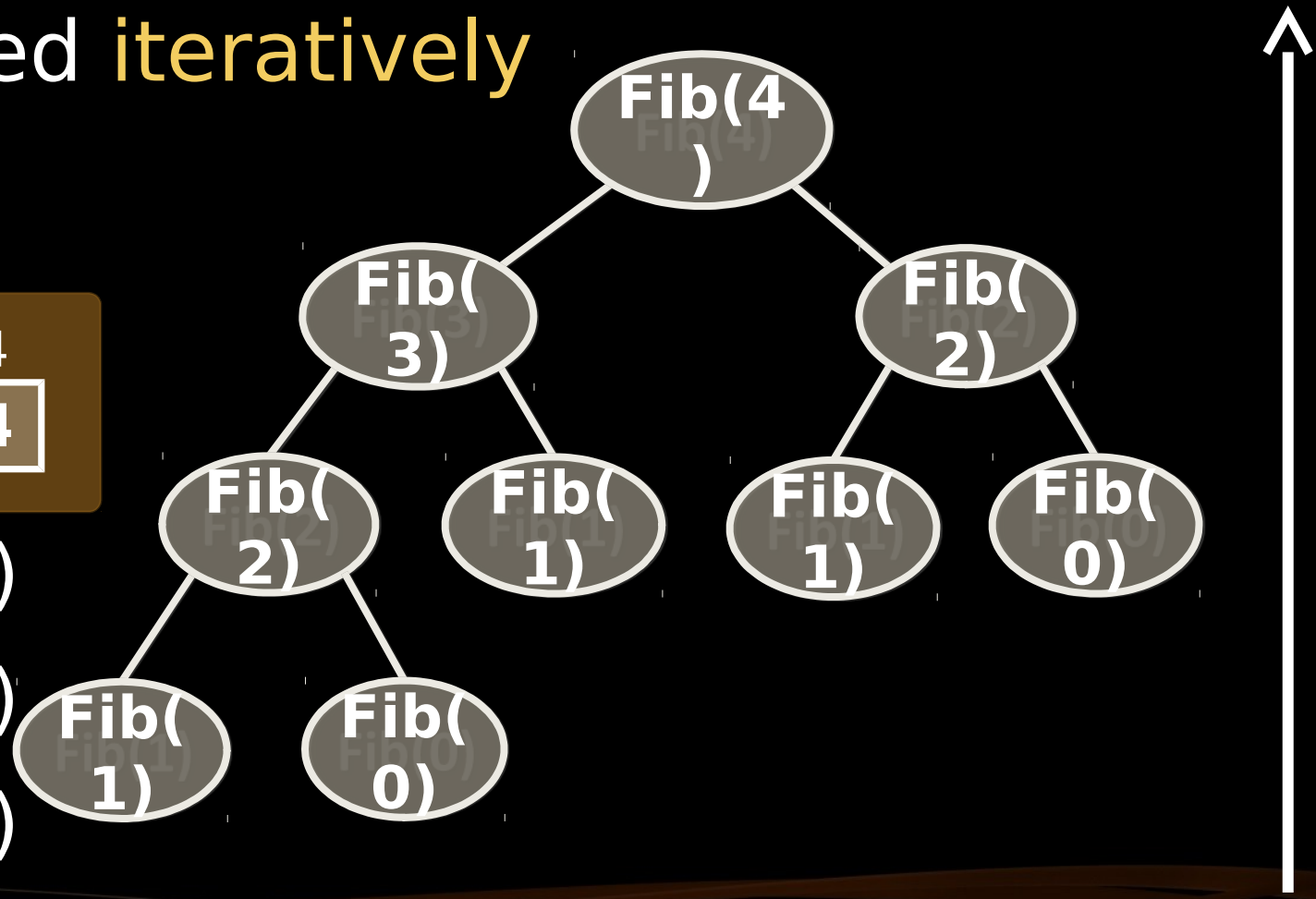
- Top-down approach in dynamic programming
- Typically implemented recursively
- Example: Fibonacci
  - Fib(4) → Fib(3), Fib(2)
  - Fib(3) → Fib(2), Fib(1)
  - Fib(2) → Fib(1), Fib(0)
  - Fib(1), Fib(0)

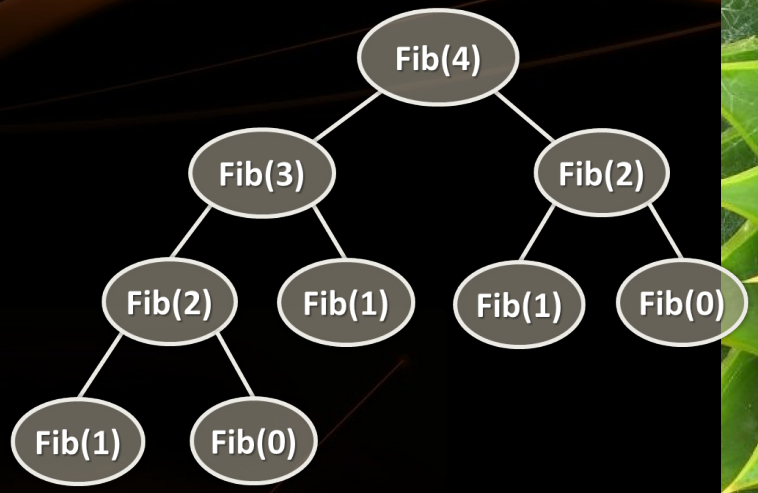


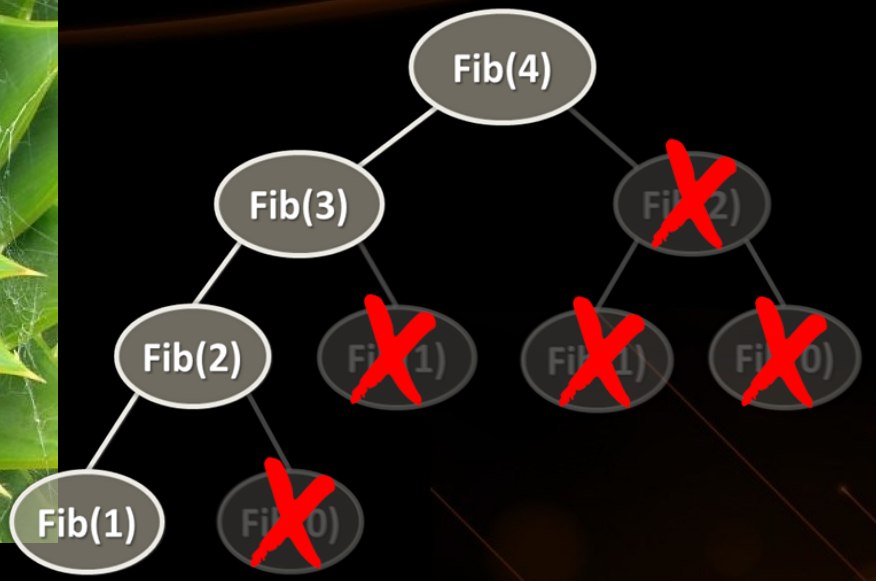
Dynamic Programming: Bottom-Up

- Bottom-up approach in dynamic programming
- Typically implemented iteratively
- Example: Fibonacci
  - Fib(0)
  - Fib(1)
  - Fib(0), Fib(1) → Fib(2)
  - Fib(1), Fib(2) → Fib(3)
  - Fib(2), Fib(3) → Fib(4)

0	1	2	3	4
0	1	1	2	4







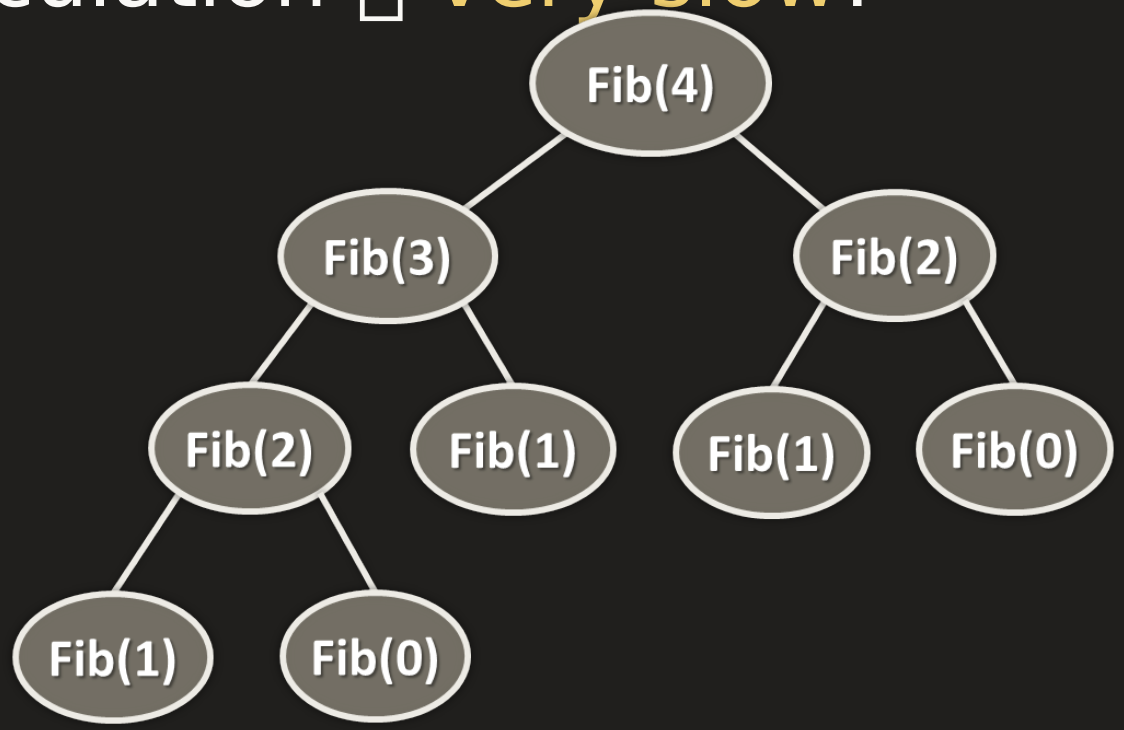
Fibonacci Calculation

- From "Divide-and-Conquer" to Dynamic Programming and Memoization

Fibonacci and Divide-and-Conquer

- Find the n^{th} Fibonacci number using recursion directly
  - Divide-and-conquer calculation very slow!

```
decimal Fibonacci(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return
    Fibonacci(n - 1) +
    Fibonacci(n - 2);
}
```



Fibonacci and DP: Top-Down (Memoization)

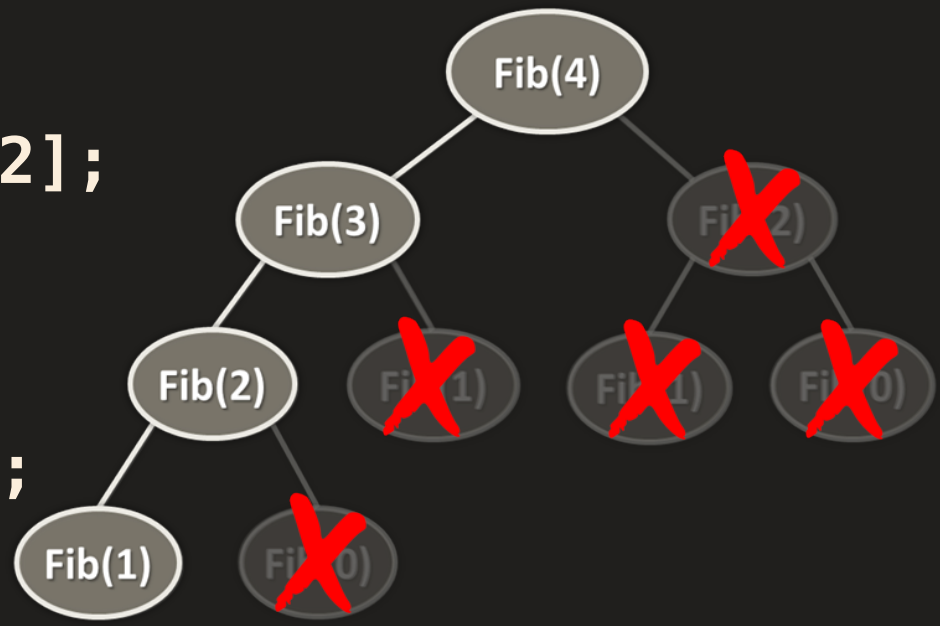
- Save each calculated value in a table for further use
  - This saves a lot of useless recursive calculations!
  - Also known as top-down calculation with memoization

```
decimal Fibonacci(int n)
{
  if (memo[n] != 0) return memo[n];
  if (n == 0) return 0;
  if (n == 1) return 1;
  memo[n] = Fibonacci(n - 1) +
    Fibonacci(n - 2);
  return memo[n];
}
```

Fibonacci and DP: Bottom-Up

- Calculating Fibonacci in bottom-up order, iteratively:

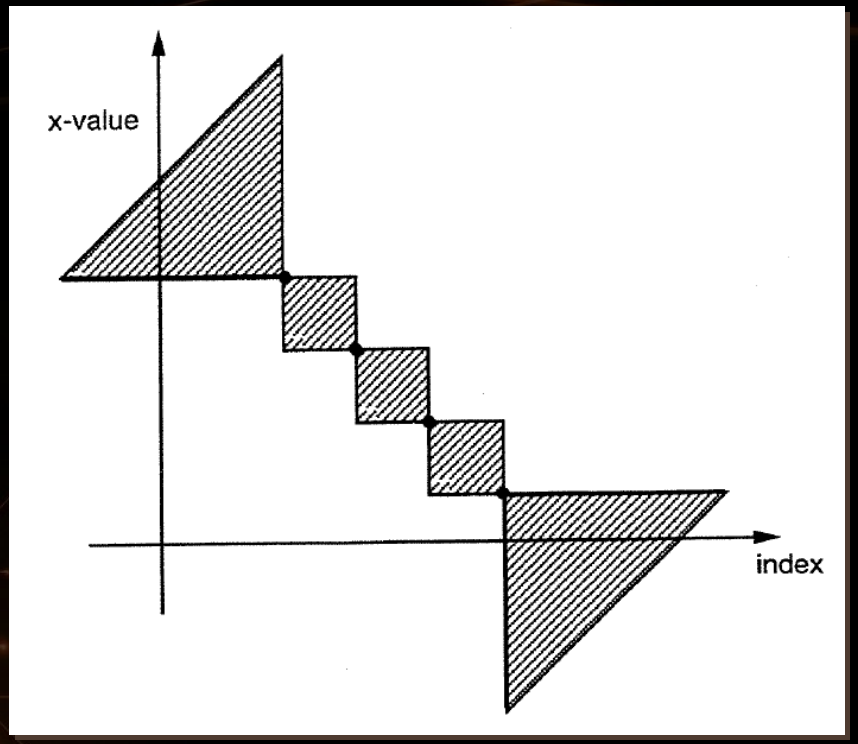
```
decimal Fibonacci(int n)
{
  decimal[] fib = new decimal[n+2];
  fib[0] = 0;
  fib[1] = 1;
  for (int i=2; i<=n; i++)
    fib[i] = fib[i-1] + fib[i-2];
  return fib[n];
}
```

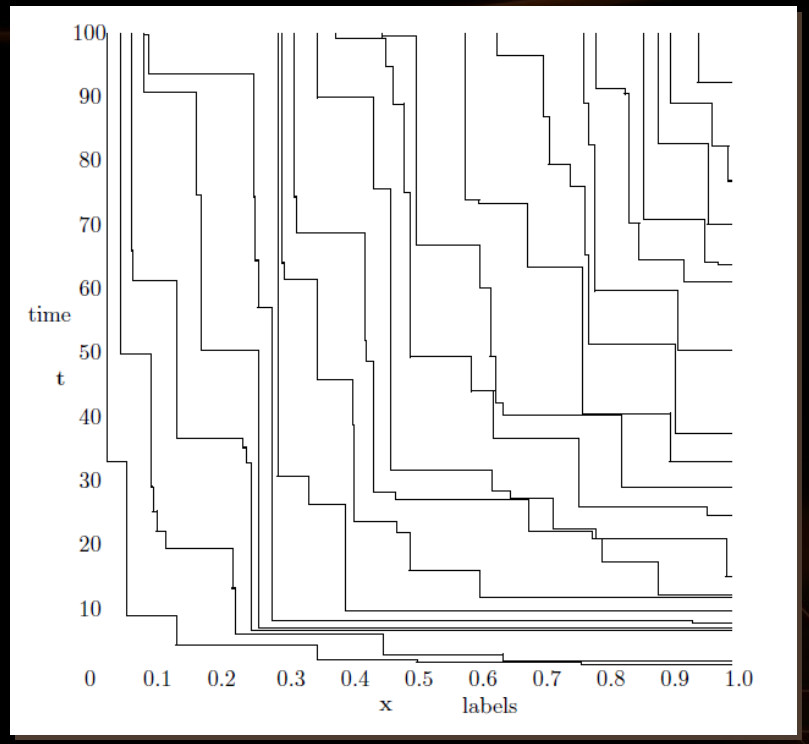


0	1	2	3	4	5	6	…	i^{th}	…	n^{th}
0	1	1	2	3	5	8	…	$F_{i-1} + F_{i-2}$	…	…

Compare Fibonacci Solutions

- Recursive Fibonacci (divide-and-conquer, no memoization)
  - Complexity: ~ $O(1.6^n)$
- Top-down dynamic programming (recursive with memorization)
  - Complexity: ~ $O(n)$
- Bottom-up dynamic programming (iterative)
  - Complexity: ~ $O(n)$
- If we want to find the 36^{th} Fibonacci number:
  - Recursive solution takes ~ 48 315 633 steps





Longest Increasing Subsequence (LIS)

Longest Increasing Subsequence (LIS)

- LIS (longest increasing subsequence) problem
  - Goal: find the largest subsequence of increasing numbers within a given sequence
  - This subsequence is not necessarily contiguous, or unique
- Example:
  - {7, **3**, **5**, 8, -1, **6**, **7**} → {3, 5, 6, 7}
- The LIS problem is solvable in O(n * log n) time
- Let's review a simple DP algorithm with complexity of $O(n^2)$

LIS – Solution with Dynamic Programming

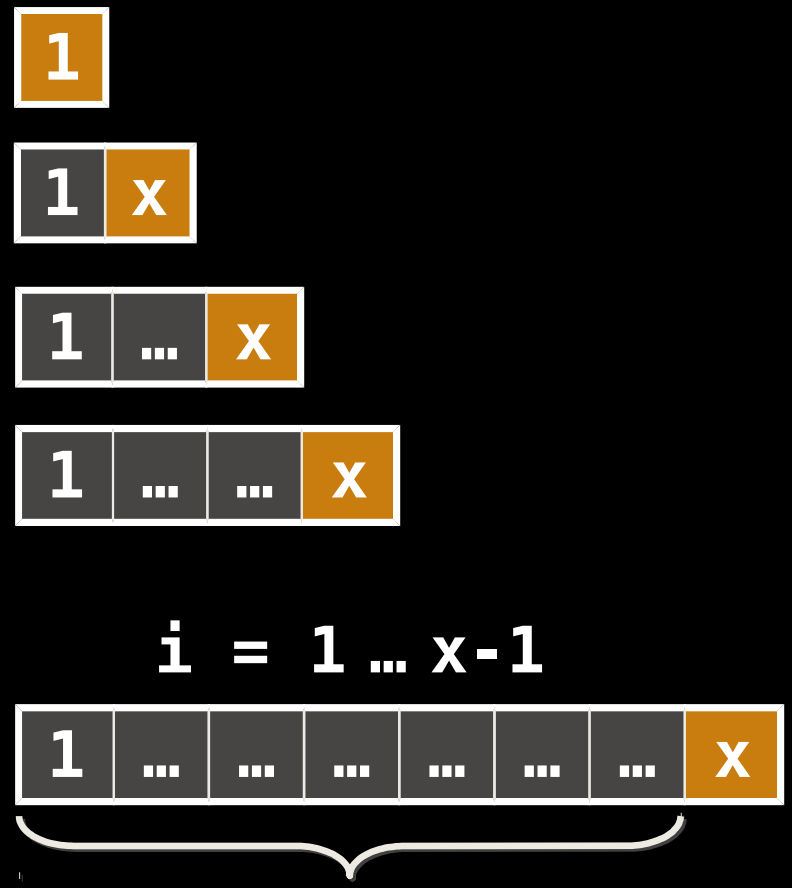
- Let's **seq[]** = {3, 14, 5, 12, 15, 7, 8, 9, 11, 10, 1}
- Let's **len[x]** = length of the LIS ending at the element **seq[x]**

index	0	1	2	3	4	5	6	7	8	9	10
seq[]	**3**	**14**	**5**	**12**	**15**	**7**	**8**	**9**	**11**	**10**	**1**
len[]	1	2	2	3	4	3	4	5	6	6	1
LIS	{**3**}	{3,**14**}	{3,**5**}	{3,5,**12**}	{3,5,12,**15**}	{3,5,**7**}	{3,5,7,**8**}	{3,5,7,8,**9**}	{3,5,7,8,9,**11**}	{3,5,7,8,9,**10**}	{**1**}

- We have optimal substructure and a recursive formula:
  - **len[x] = 1 + max(len[i])**
    for **i** ∈ [**0** … **x-1**], where **seq[i] < seq[x]**

LIS: Bottom-Up Calculation

- Bottom-up calculation the LIS length:
  - `len[0] = 1`
  - `len[0] □ len[1]`
  - `len[0…1] □ len[2]`
  - `len[0…2] □ len[3]`
  - …
  - `len[0…n-1] □ len[n]`



Calculating LIS – Source Code


```
int[] seq = { 3, 4, 8, 1, 2, 4, 32, 6, 2, 5, 33, 4, 38,
22 };
int[] len = new int[seq.Length];
for (int x = 0; x < seq.Length; x++)
{
  len[x] = 1;
  for (int i = 0; i <= x - 1; i++)
    if (seq[i] < seq[x] && len[i] + 1 > len[x])
      len[x] = 1 + len[i];
}
```

index	0	1	2	3	4	5	6	7	8	9	10
seq[]	**3**	**14**	**5**	**12**	**15**	**7**	**8**	**9**	**11**	**10**	**1**
len[]	1	2	2	3	4	3	4	5	6	6	1

LIS – Restoring the Sequence

- How to restore the LIS from the calculated `len[]` values?
  - We can keep `prev[x]` to hold the predecessor of `seq[x]`

index	0	1	2	3	4	5	6	7	8	9	10
seq[]	**3**	**14**	**5**	**12**	**15**	**7**	**8**	**9**	**11**	**10**	**1**
len[]	1	2	2	3	4	3	4	5	6	6	1
prev[]	-1	0	0	2	3	2	5	6	7	7	-1
LIS	{**3**}	{3,**14**}	{3,**5**}	{3,5,**12**}	{3,5,12,**15**}	{3,5,**7**}	{3,5,7,**8**}	{3,5,7,8,**9**}	{3,5,7,8,9,**11**}	{3,5,7,8,9,**10**}	{**1**}

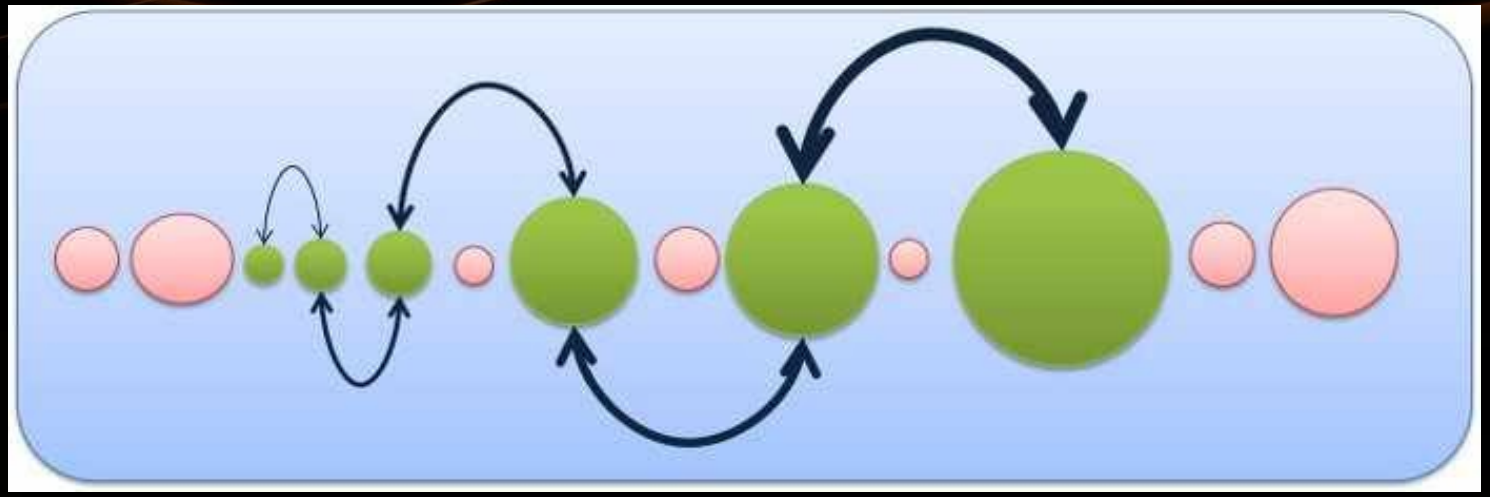
Calculating LIS with Previous – Source Code


```
int maxLen = 0;
int lastIndex = -1;
for (int x = 0; x < seq.Length; x++)
{
  len[x] = 1;
  prev[x] = -1;
  for (int i = 0; i < x; i++)
    if ((seq[i] < seq[x]) && (len[i] + 1 > len[x]))
    {
      len[x] = len[i] + 1;
      prev[x] = i;
    }
  if (len[x] > maxLen)
  {
    maxLen = len[x];
    lastIndex = x;
  }
}
```

Restoring LIS Elements – Source Code


```
int[] RestoreLIS(int[] seq, int[] prev, int lastIndex)
{
  var longestSeq = new List<int>();
  while (lastIndex != -1)
  {
    longestSeq.Add(sequence[lastIndex]);
    lastIndex = prev[lastIndex];
  }

  longestSeq.Reverse();

  return longestSeq.ToArray();
}
```

SOFTWARE UNIVERSITY FOUNDATION



Longest Increasing Subsequence (LIS)

Live Coding (Lab)

2	6	1	8	9	4	2
1	8	0	3	5	6	7
3	4	8	7	2	1	8
0	9	2	8	1	7	9
2	7	1	9	7	8	2
4	5	6	1	2	5	6
9	3	5	2	8	1	9
2	3	4	1	7	2	8

2	6	1	8	9	4	2
1	8	0	3	5	6	7
3	4	8	7	2	1	8
0	9	2	8	1	7	9
2	7	1	9	7	8	2
4	5	6	1	2	5	6
9	3	5	2	8	1	9
2	3	4	1	7	2	8

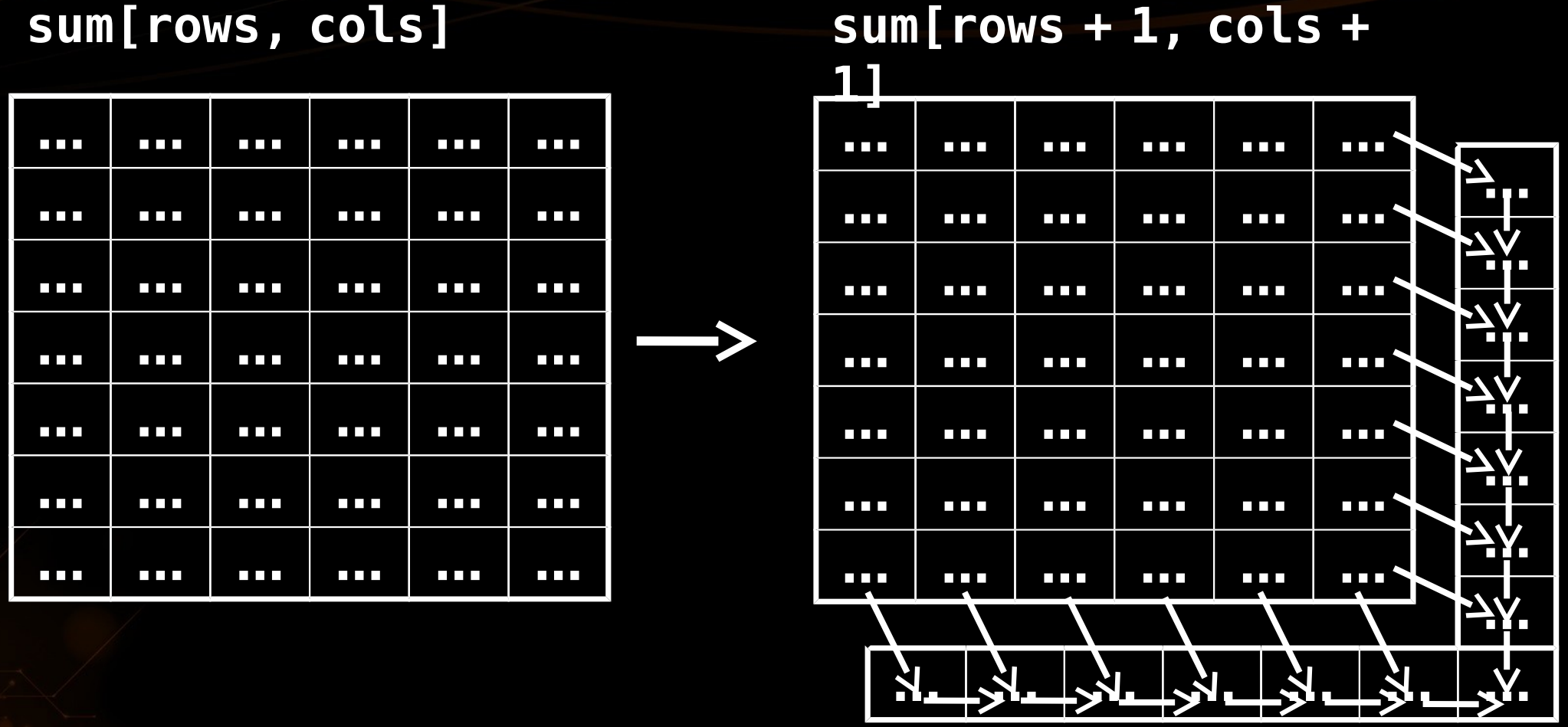
"Move Down / Right Sum" Problem

"Move Down / Right Sum" Problem

- You are given a matrix of numbers
  - Find the largest sum of cells
  - Starting from the top left corner
  - Ending at the bottom right corner
  - Going down or right at each step
- Dynamic programming solution:
  - `sum[row, col] = cell[row, col] + max(sum[row-1, col], sum[row, col-1])`

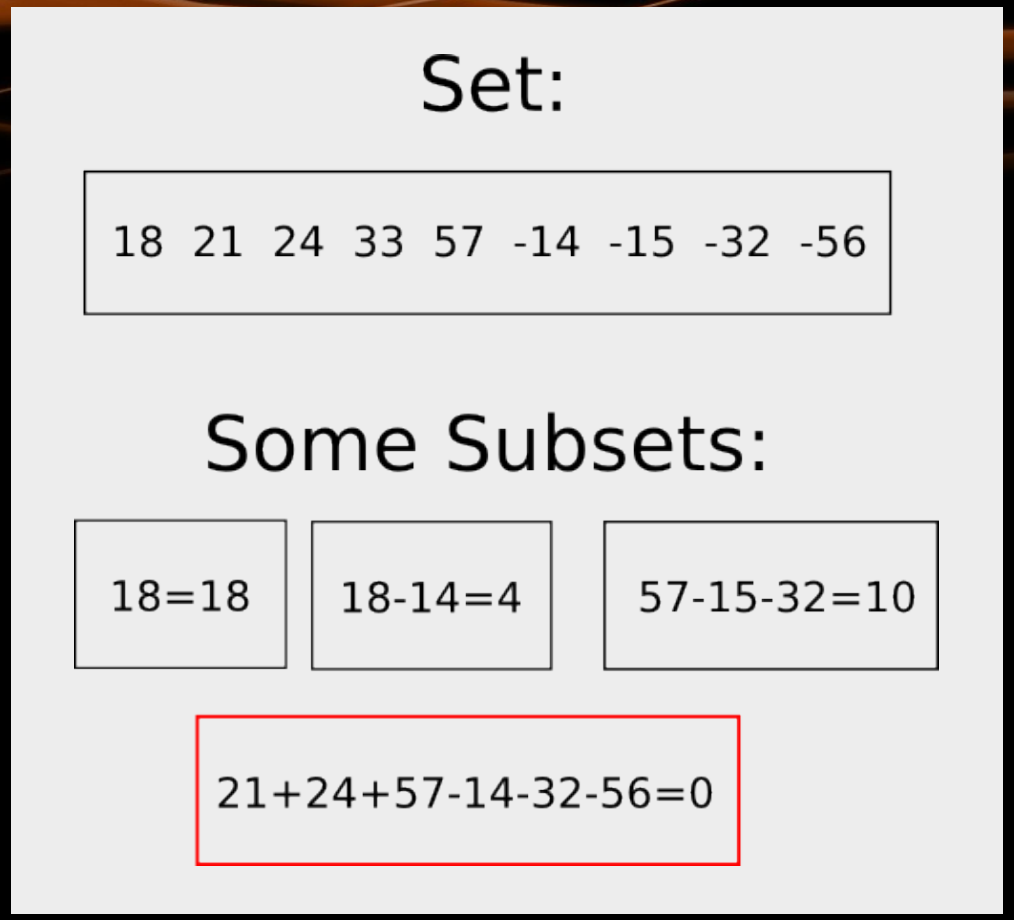
2	**6**	1	8	9	4	2
1	**8**	0	3	5	6	7
3	**4**	**8**	**7**	2	1	8
0	9	2	**8**	1	7	9
2	7	1	**9**	**7**	**8**	2
4	5	6	1	2	**5**	**6**
9	3	5	2	8	1	**9**
2	3	4	1	7	2	**8**

Move Down / Right: Calculation Process



"Move Down / Right Sum" – Solution

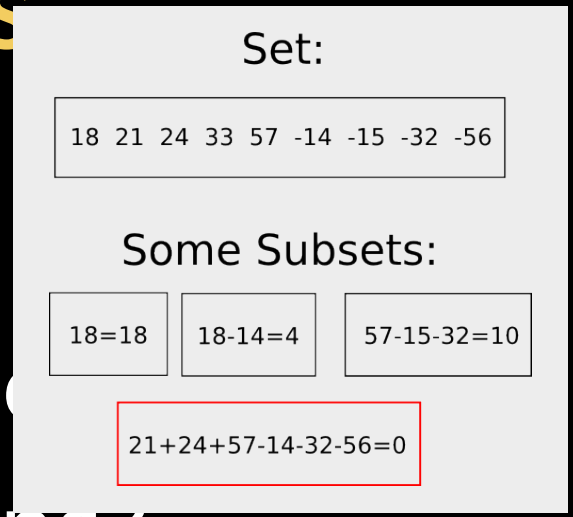

```
for (int row = 0; row < rowsCount; row++)
  for (int col = 0; col < colsCount; col++)
  {
    long maxPrevCell = long.MinValue;
    if (col > 0 && sum[row, col - 1] > maxPrevCell)
      maxPrevCell = sum[row, col - 1];
    if (row > 0 && sum[row - 1, col] > maxPrevCell)
      maxPrevCell = sum[row - 1, col];
    sum[row, col] = cells[row, col];
    if (maxPrevCell != long.MinValue)
      sum[row, col] += maxPrevCell;
  }
```



The "Subset Sum" Problem

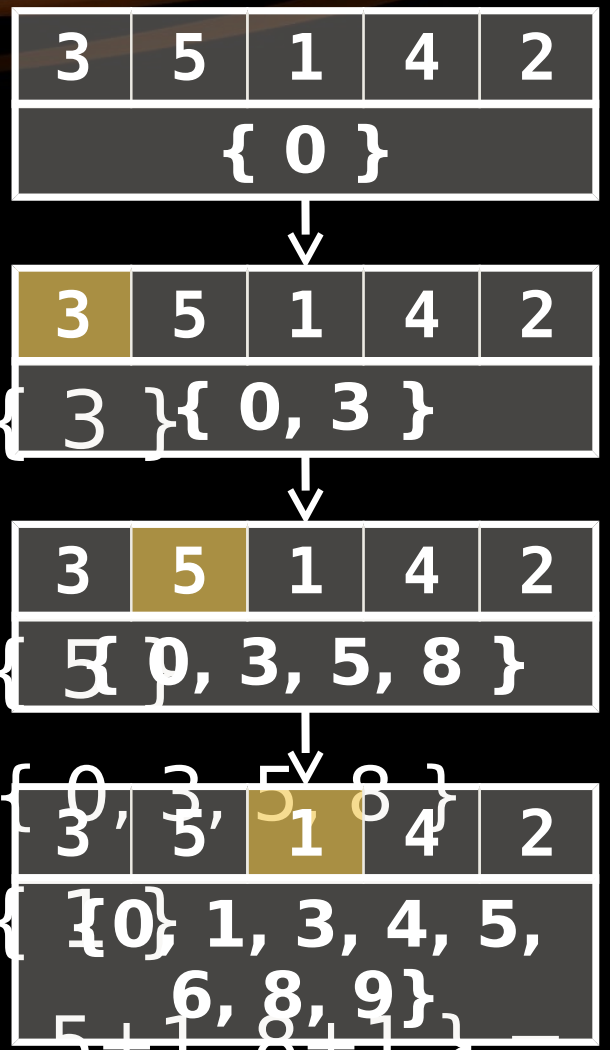
Subset Sum Problem and Its Variations

- Subset sum problem (zero subset sum problem)
  - Given a set of integers, find a non-empty subset whose sum 0
    - E.g. {8, **3**, -50, **1**, **-2**, -1, 15, **-2**} → {3, 1, -2, -2}
  - Given a set of integers and an integer **S**, find a subset whose sum is **S**
    - E.g. {8, **3**, 2, **1**, **12**, 1}, **S**=16 → {3, 1, 12}
- Given a set of integers, find all possible
- Can you split given sum into set of coins?



Subset Sum Problem (No Repeats)

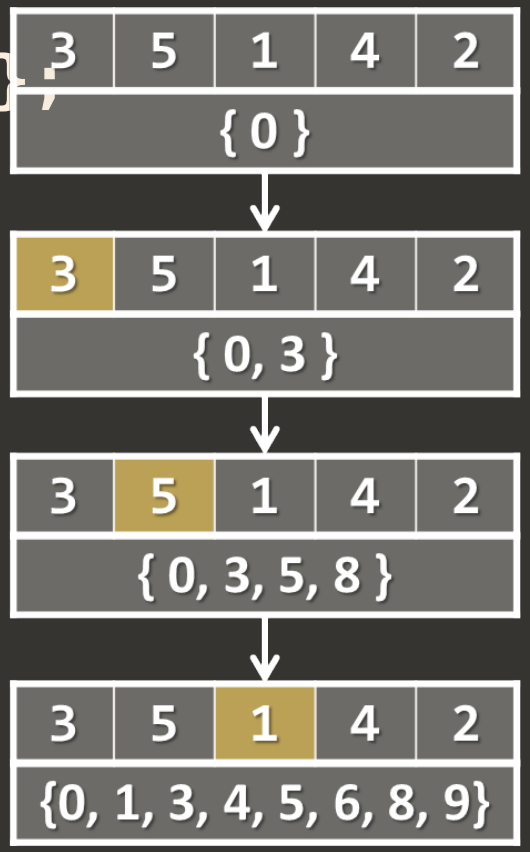
- Solving the subset sum problem:
  - **nums** = { 3, 5, 1, 4, 2 }, **targetSum** = 6
- Start with **possibleSums** = { 0 }
- Step 1: obtain all possible sums ending at { 3 }
  - **possibleSums** = { 0 } ∪ { 0+3 } = { 0, 3 }
- Step 2: obtain all possible sums ending at { 5 }
  - **possibleSums** = { 0, 3 } ∪ { 0+5, 3+5 } = { 0, 3, 5, 8 }
- Step 3: obtain all possible sums ending at { 1 }
  - **possibleSums** = { 0, 3, 5, 8 } ∪ { 0+1, 3+1, 5+1, 8+1 } = {0, 1, 3, 4, 5, 6, 8, 9}



Subset Sum Problem (No Repeats)

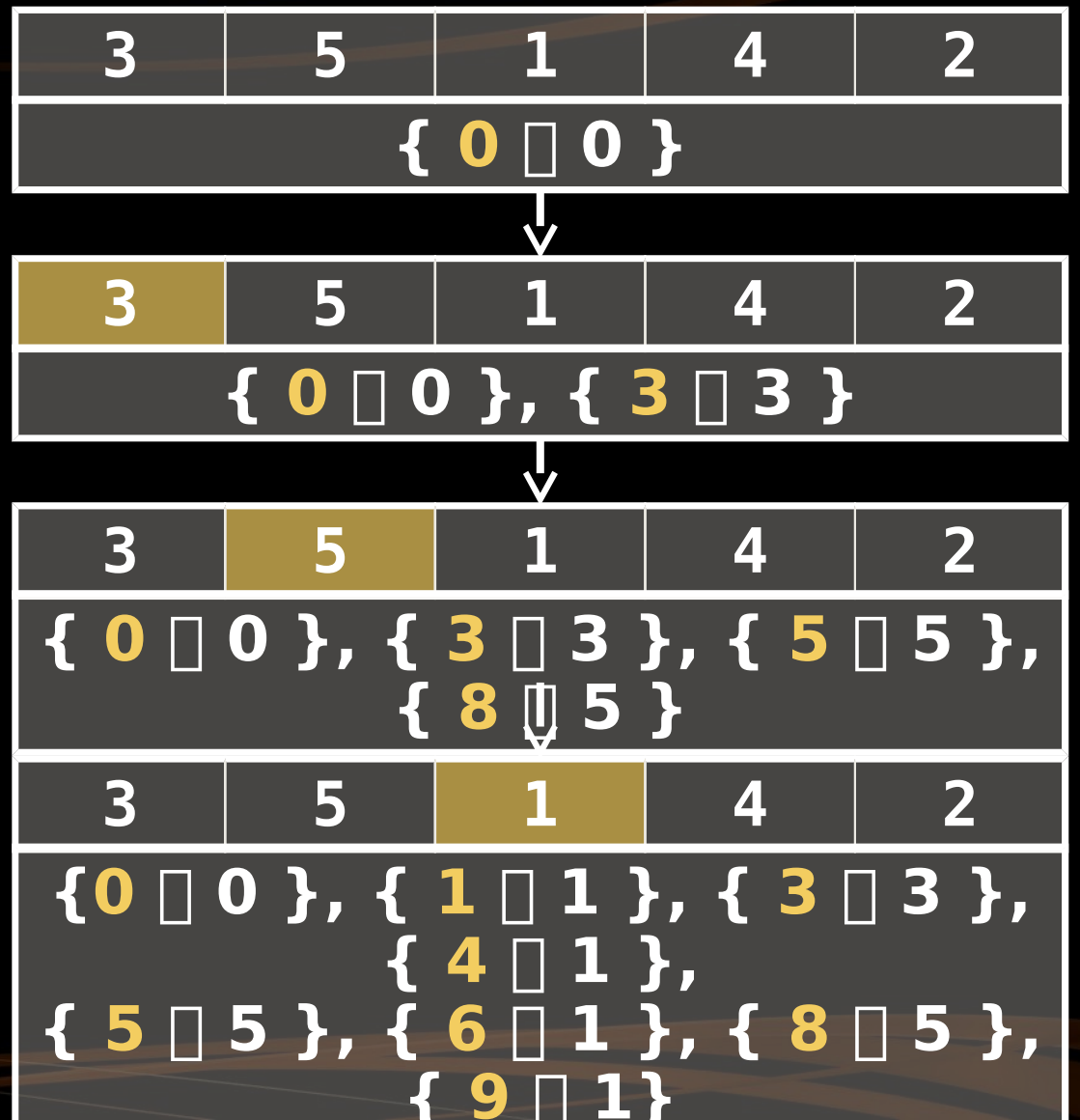
SOFTWARE UNIVERSITY FOUNDATION

```
static ISet<int> CalcPossibleSumsSet(int[] nums, int
targetSum)
{
  var possibleSums = new HashSet<int>() { 0 };
  for (int i = 0; i < nums.Length; i++)
  {
    var newSums = new HashSet<int>();
    foreach (var sum in possibleSums)
    {
      int newSum = sum + nums[i];
      newSums.Add(newSum);
    }
    possibleSums.UnionWith(newSums);
  }
  return possibleSums;
}
```



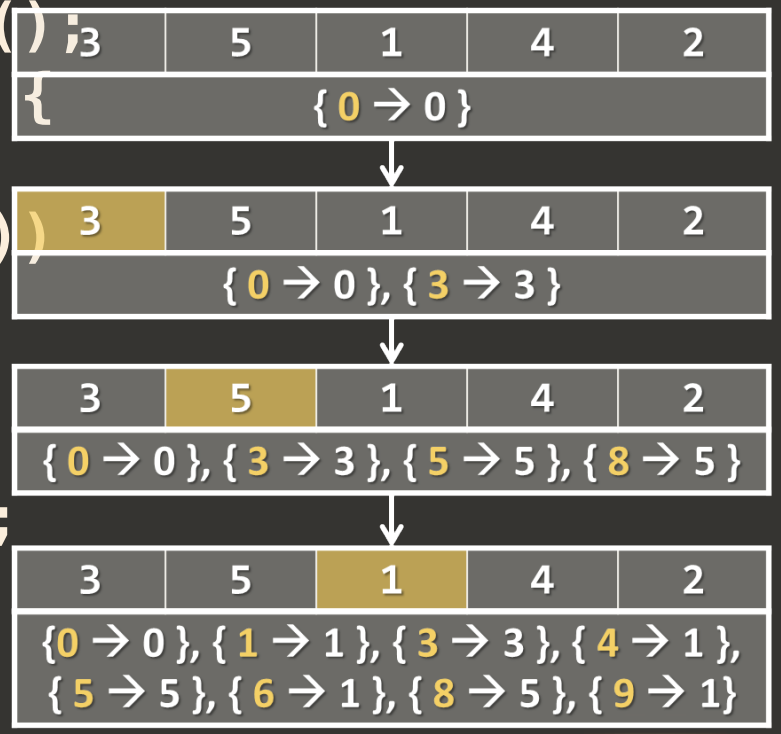
Subset Sum: How to Recover the Subset?

- Keep for each obtained sum in **possibleSums**, how it is obtained
- Use a dictionary instead of set:
  - **possibleSums[*s*] = *num***
  - The sum ***s*** is obtained by adding ***num*** to some previously obtained subset sum



Subset Sum (No Repeats + Subset Recovery)

```
static IDictionary<int, int> CalcPossibleSums(
    int[] nums, int targetSum) {
  var possibleSums = new Dictionary<int, int>();
  possibleSums.Add(0, 0);
  for (int i = 0; i < nums.Length; i++) {
    var newSums = new Dictionary<int, int>();
    foreach (var sum in possibleSums.Keys) {
      int newSum = sum + nums[i];
      if (!possibleSums.ContainsKey(newSum))
        newSums.Add(newSum, nums[i]);
    }
    foreach (var sum in newSums)
      possibleSums.Add(sum.Key, sum.Value);
  }
  return possibleSums;
}
```



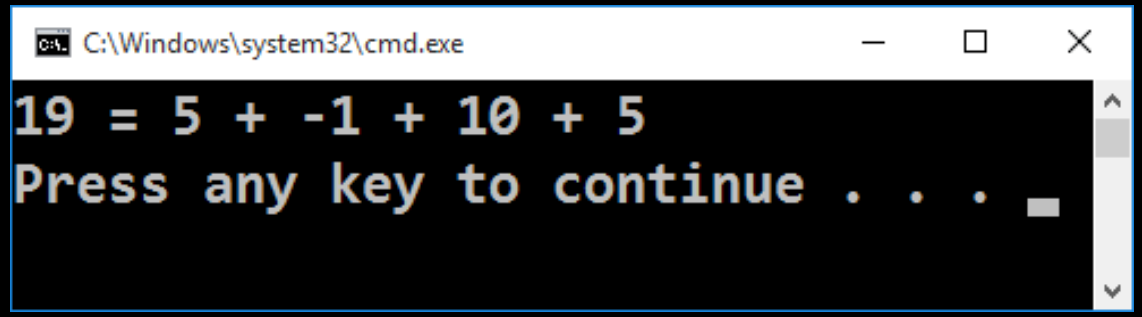
Subset Sum (No Repeats): Subset Recovery


```
static List<int> FindSubset(int[] nums, int targetSum,
  IDictionary<int, int> possibleSums)
{
  var subset = new List<int>();
  while (targetSum > 0)
  {
    var lastNum = possibleSums[targetSum];
    subset.Add(lastNum);
    targetSum -= lastNum;
  }

  subset.Reverse();

  return subset;
}
```

{ 9 → 1} → { 8 → 5 } → { 5 → 5 } → 0

```
int[] nums = { 3, 5, -1, 10, 5, 7 };
int targetSum = 19;

var possibleSums = CalcPossibleSums(nums, targetSum);
```

```
var subset = FindSubset(nums, targetSum, possibleSums);
Console.Write(targetSum + " = ");
Console.WriteLine(String.Join(" + ", subset));
```



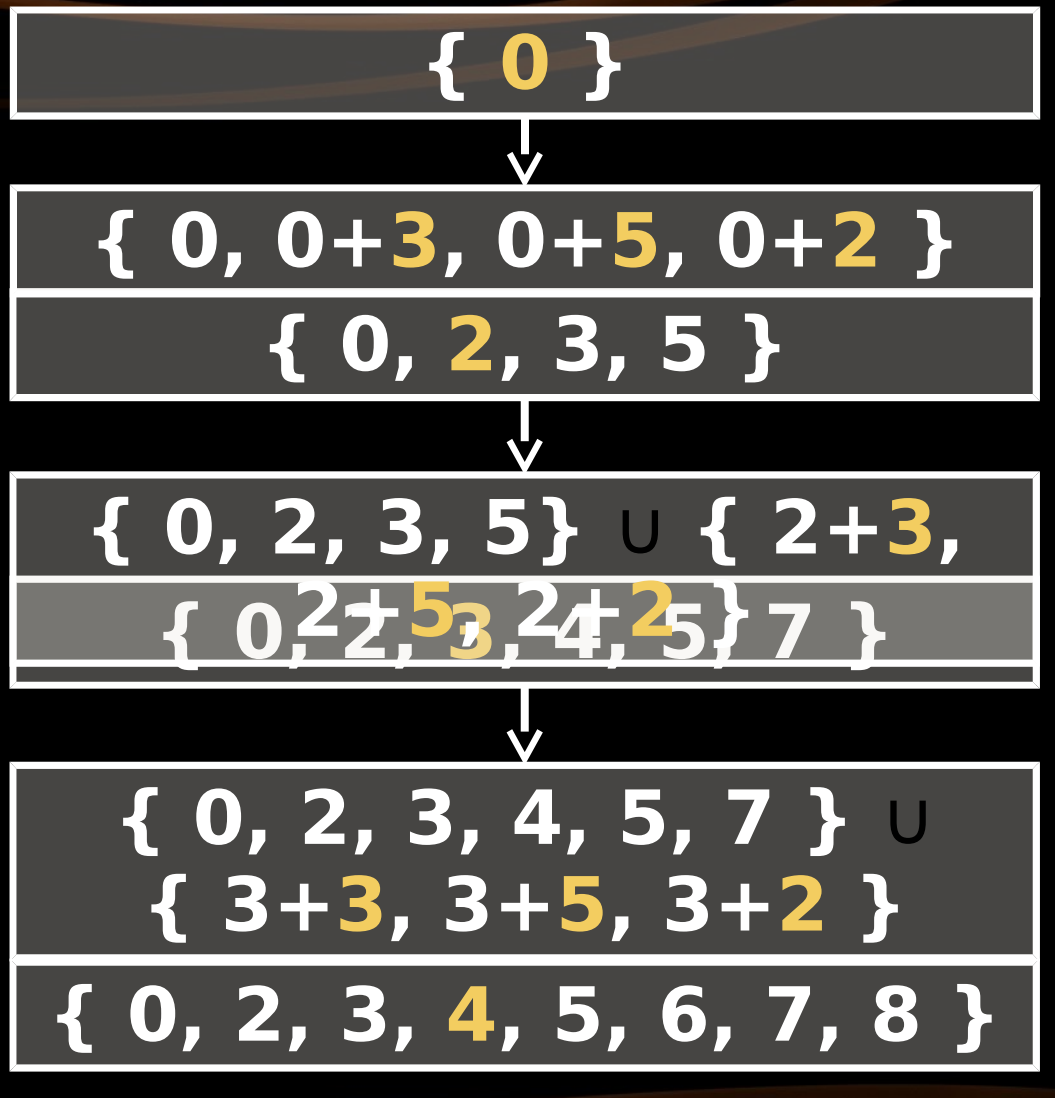
Subset Sum (with No Repeats)

Live Demo

Subset Sum Problem (With Repeats)

- Solving the subset sum problem (BFS-based algorithm):
  - **nums** = { 3, 5, 2 }, **targetSum** = 6

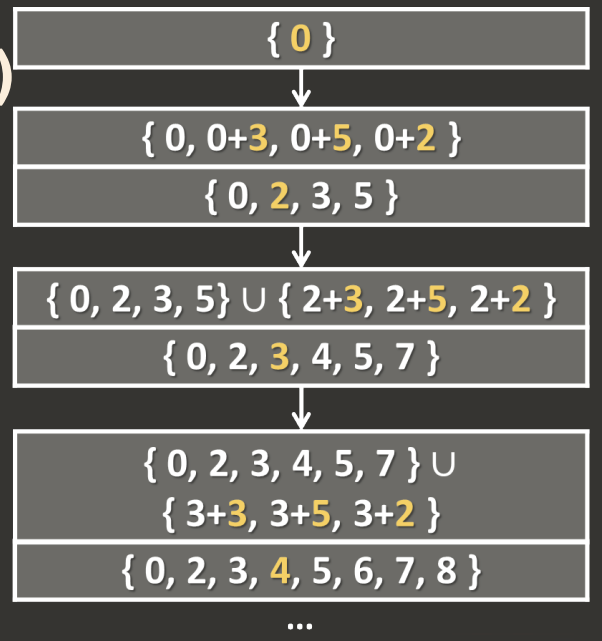
```
possibleSums = { 0 };
for i = 0 … targetSum
{
   Append to possibleSums[i] all
   numbers from nums[]
}
```



Subset Sum (With Repeats)

SOFTWARE UNIVERSITY FOUNDATION

```
static bool[] CalcPossibleSums(int[] nums, int targetSum)
{
  var possible = new bool[targetSum + 1];
  possible[0] = true;
  for (int sum = 0; sum < possible.Length; sum++)
    if (possible[sum])
      for (int i = 0; i < nums.Length; i++)
      {
        int newSum = sum + nums[i];
        if (newSum <= targetSum)
          possible[newSum] = true;
      }
  return possible;
}
```

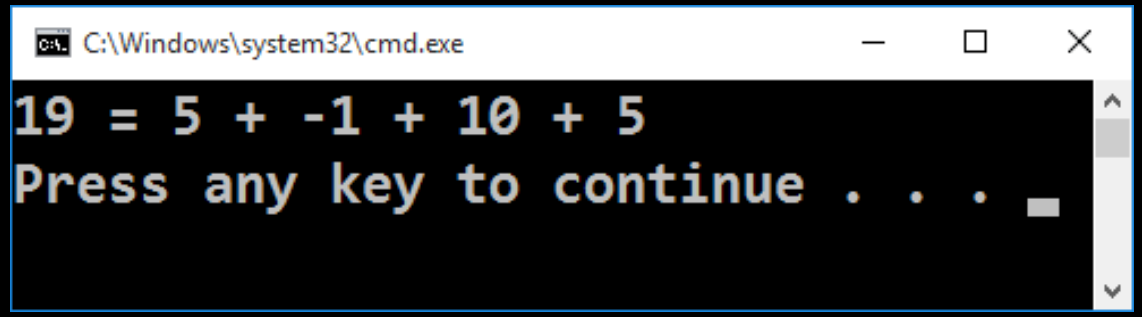


Subset Sum (With Repeats): Recovery


```
private static IEnumerable<int> FindSubset(
  int[] nums, int targetSum, bool[] possibleSums)
{
  var subset = new List<int>();
  while (targetSum > 0)
    for (int i = 0; i < nums.Length; i++)
    {
      int newSum = targetSum - nums[i];
      if (newSum >= 0 && possibleSums[newSum])
      {
        targetSum = newSum;
        subset.Add(nums[i]);
      }
    }
  return subset;
}
```

```
int[] nums = { 3, 5, -1, 10, 5, 7 };
int targetSum = 19;

var possibleSums = CalcPossibleSums(nums, targetSum);
```

```
var subset = FindSubset(nums, targetSum, possibleSums);
Console.Write(targetSum + " = ");
Console.WriteLine(String.Join(" + ", subset));
```



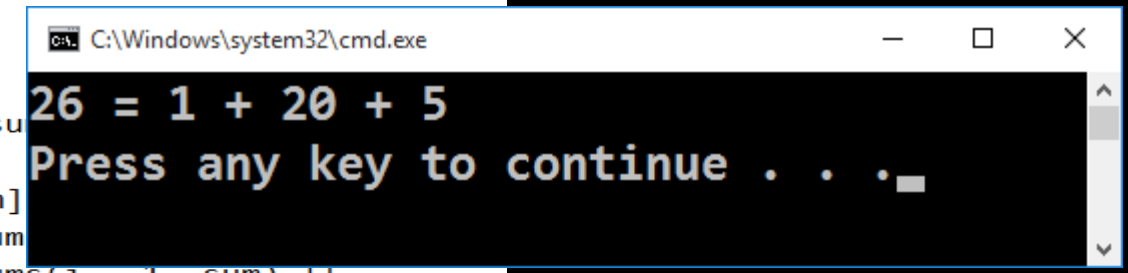
Subset Sum (With Repeats)

Live Demo

Subset Sums (No Repeat, Recursive)

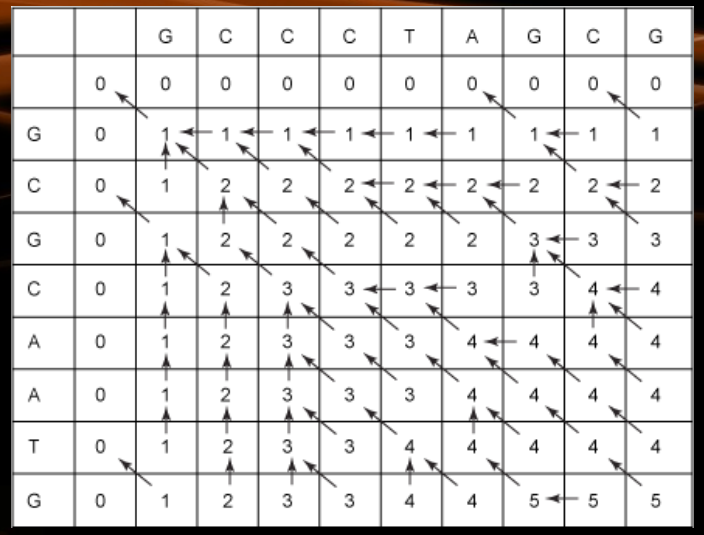
- Another solution to the "subset sums" problem
  - Let **possibleSum[*i*, *s*]** is **true** if the sum ***s*** can be obtained by summing a subset of **nums[0 … *i*]**
  - The following formulas are valid:
    - **possibleSum[*i*, *s*]** == **false** when *i* < 0 or *s* < 0
    - **possibleSum[*i*, *s*]** == **true** when **nums[i]** == **sum** or **possibleSum[*i-1*, *s*]** or **possibleSum[*i-1*, *s* – nums[*i*]]**

```
static bool CalcPossibleSums(int i, int sum)
{
    if (sum < 0 || i < 0)
    {
        return false;
    }

    if (!isCalculated[i, su
    {
        possibleSum[i, sum]
            (nums[i] == sum
            CalcPossibleSums(i - 1, sum) ||
            CalcPossibleSums(i - 1, sum - nums[i]);
        isCalculated[i, sum] = true;
    }

    return possibleSum[i, sum];
}
```

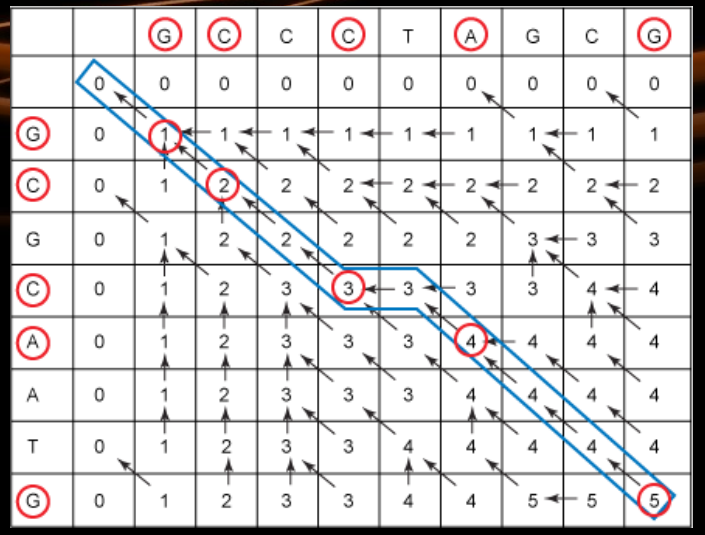


Subset Sum (With Matrix)

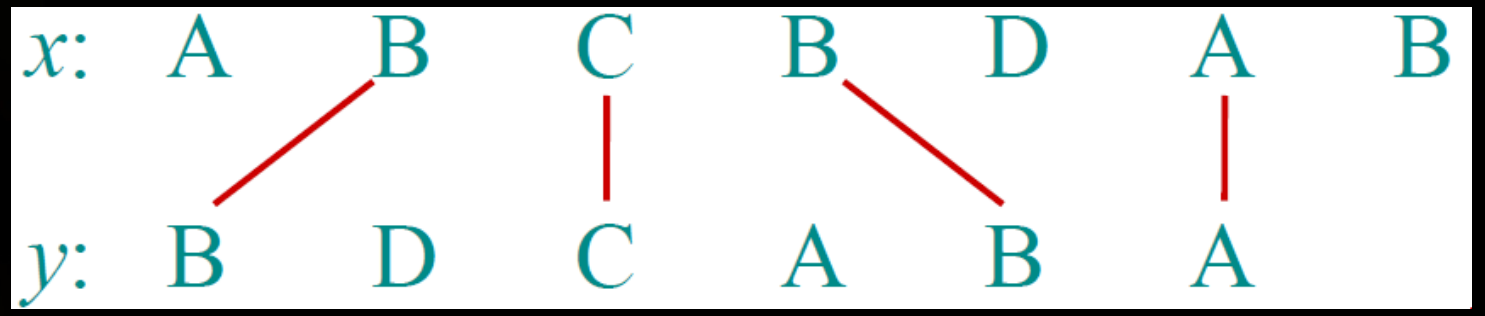
Live Demo



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

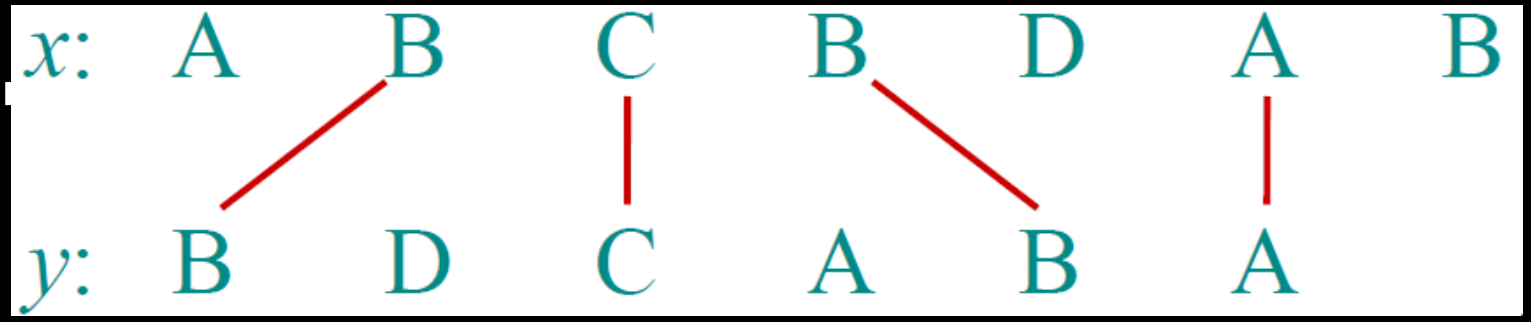


Longest Common Subsequence (LCS)

- A Recursive DP Approach

Longest Common Subsequence (LCS)

- Longest common subsequence (LCS) problem:
  - Given two sequences **x[1 … m]** and **y[1 … n]**
  - Find a longest common subsequence (LCS) to them both
- Example:
  - x = "A**BCB**D**A**B"
  - y = "**B**D**C**A**BA**"
  - LCS = "**BCBA**"



LCS – Recursive Approach

- S_1 = GCCCTAGCG, S_2 = GCGCAATG
  - Let C_1 = the right-most character of S_1 (C_1 = G)
  - Let C_2 = the right-most character of S_2 (C_2 = G)
  - Let S_1' = S_1 with C_1 "chopped-off" (S_1' = GCCCTAGC)
  - Let S_2' = S_2 with C_2 "chopped-off" (S_2' = GCGCAAT)
- There are three recursive sub-problems:
  - L_1 = LCS(S_1', S_2)
  - L_2 = LCS(S_1, S_2')
  - L_3 = LCS(S_1', S_2')

LCS – Recursive Formula

- Let `lcs[x, y]` is the longest common subsequence of
  $S_1[0 \dots x]$ and $S_2[0 \dots y]$
- LCS has the following recursive properties:

```
lcs[-1, y] = 0
lcs[x, -1] = 0
lcs[x, y] = max(
   lcs[x-1, y],
   lcs[x, y-1],
   lcs[x-1, y-1] when S1[x] == S2[y]
)
```

Calculating the LCS Table

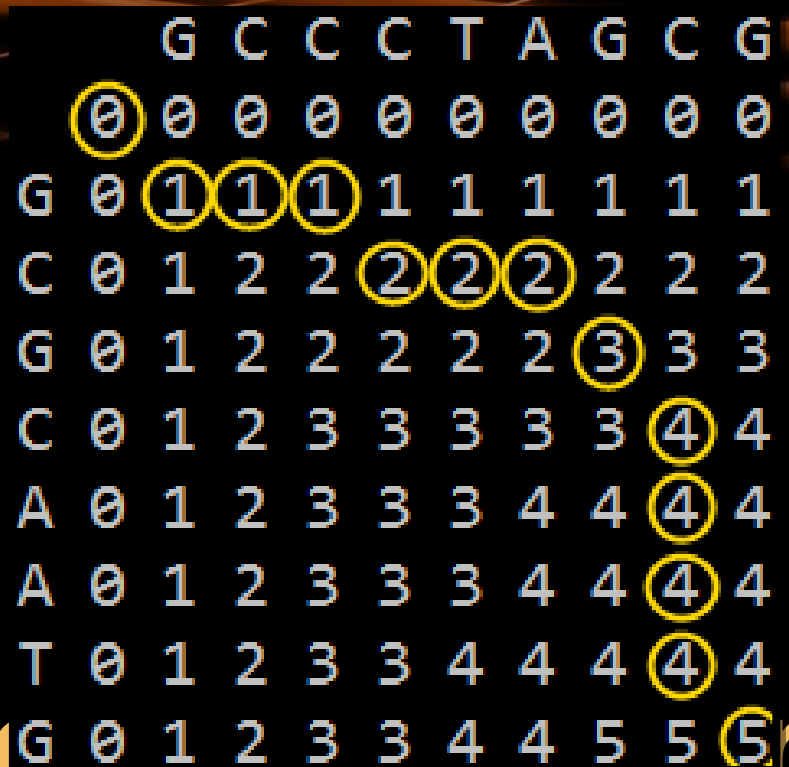
```
static int CalcLCS(int x, int y) {
  if (x < 0 || y < 0)
    return 0;
  if (lcs[x, y] == NOT_CALCULATED)  {
    int lcsFirstMinusOne = CalcLCS(x - 1, y);
    int lcsSecondMinusOne = CalcLCS(x, y - 1);
    lcs[x, y] = Math.Max(
      lcsFirstMinusOne, lcsSecondMinusOne);
    if (firstStr[x] == secondStr[y]) {
      int lcsBothMinusOne = 1 + CalcLCS(x - 1, y - 1);
      lcs[x, y] = Math.Max(lcs[x, y], lcsBothMinusOne);
    }
  }
  return lcs[x, y];
}
```

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

Reconstructing the LCS Sequence

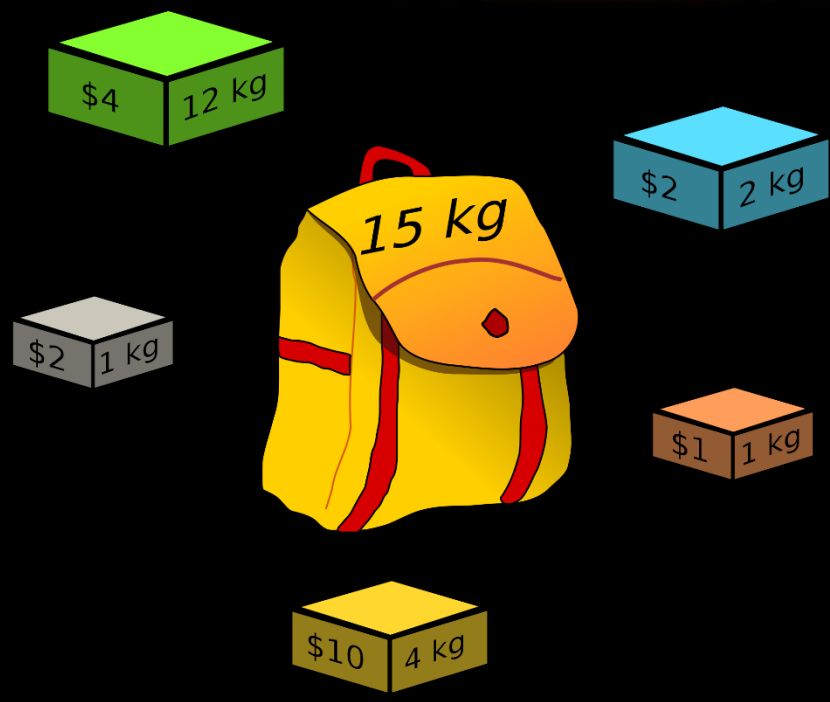
```
static string PrintLCS(int x, int y) {
  var lcsLetters = new List<char>();
  while (x >= 0 && y >= 0)
    if ((firstStr[x] == secondStr[y]) &&
      (CalcLCS(x - 1, y - 1) + 1 == lcs[x, y]))
      lcsLetters.Add(firstStr[x]);
      x--;
      y--;
    }
    else if (CalcLCS(x - 1, y) == lcs[x, y])
      x--;
    else
      y--;
  lcsLetters.Reverse();
  return string.Join("", lcsLetters);
}
```

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5



Longest Common Subsequence (LCS)

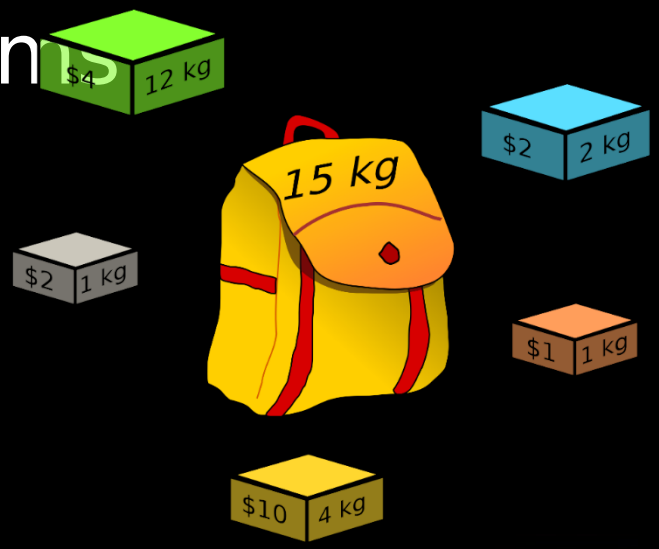
Live Coding (Lab)



The "Knapsack" Problem

The "Knapsack" Problem

- We have `n` items
  - Each item has price c_i and weight w_i
- We have a knapsack of given capacity (maximum weight)
  - We want to take maximal priced items within the knapsack capacity
- Take a subset of the items, so that:
  - The total price is maximal
  - The total weigth ≤ capacity



The "Knapsack" Problem – Example

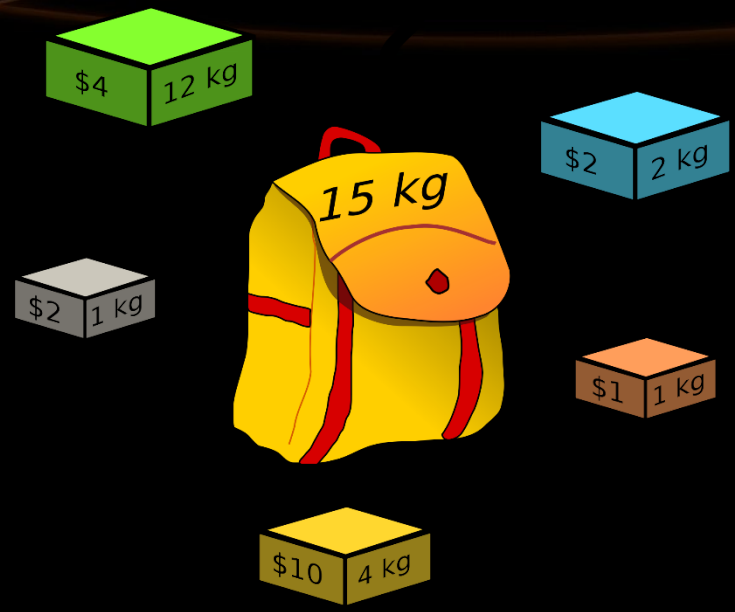
Available items:

#	Weight	Price
0	5	30
1	8	120
2	7	10
3	0	20
4	4	50
5	5	80
6	2	10

Items taken:

#	Weight	Price
1	8	120
3	0	20
5	5	80
6	2	10

SOFTWARE UNIVERSITY FOUNDATION

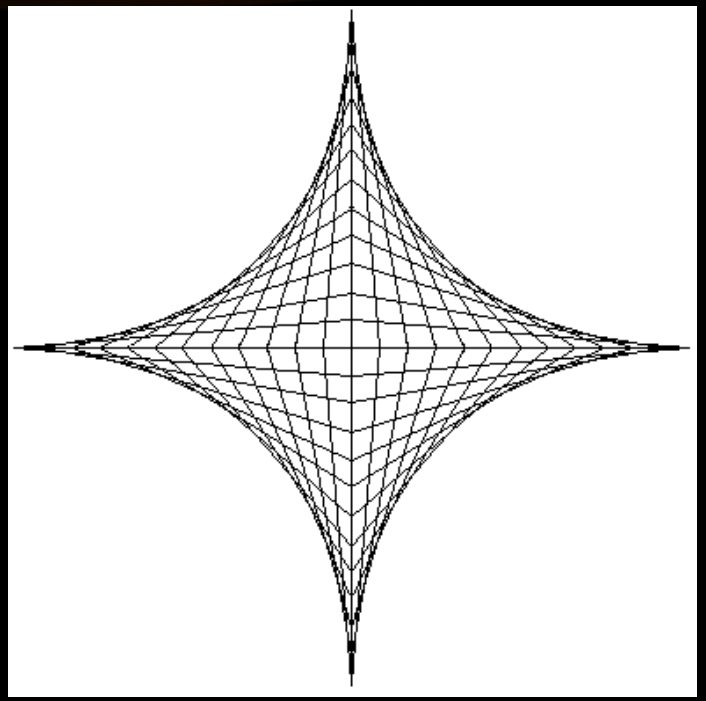


The "Knapsack" Problem

Live Coding (Lab)

Dynamic Programming Applications

- Areas
  - Bioinformatics
  - Control theory
  - Information theory
  - Operations research
  - Computer science:
    - Theory
    - Graphics
    - AI



Some Famous Dynamic Programming Algorithms

- Integer Knapsack Problem
- Unix diff for comparing two files
- Bellman–Ford algorithm for finding the shortest distance in a graph
- Floyd's All-Pairs shortest path algorithm
- Cocke-Kasami-Younger for parsing context free grammars
- en.wikipedia.org/wiki/Dynamic_programming#Algorithms_that_use_dynamic_programming

Summary

- Divide-and-conquer method for algorithm design
- Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms
- Dynamic programming is applicable when the sub-problems are dependent, that is, when sub-problems share sub-sub-problems
- Recurrent functions can be solved efficiently
- Longest increasing subsequence and Longest common subsequence problems can be solved efficiently using dynamic programming approach

Dynamic Programming

SOFTWARE UNIVERSITY FOUNDATION

Questions?

https://softuni.bg/trainings/1194/Algorithms-September-2015

License

- This course (slides, examples, labs, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with C#" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "Data Structures and Algorithms" course by Telerik Academy under CC-BY-NC-SA license

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity